O'REILLY®

TURING 图灵程序设计丛书



Java 编程思维

Think Java: How to Think Like a Computer Scientist

与AP课程对应，从编程基础知识入手，用Java代码示例诠释计算机科学概念，教读者掌握“解决问题”的思维方式

[美] Allen B. Downey Chris Mayfield 著
袁国忠 译

 中国工信出版集团  人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

译者介绍



袁国忠

自由译者；2000年起专事翻译，主译图书，偶译新闻稿、软文；出版译著40余部，其中包括《Python编程：从入门到实践》《C++ Prime Plus中文版》《CCNA学习指南》《CCNP ROUTE学习指南》《面向模式的软件架构：模式系统》《风投的选择：谁是下一个十亿美元级公司》等，总计700余万字；专事翻译前，做过两年杂志和图书编辑，从事过三年化工产品分析和开发。



图灵程序设计丛书

Java编程思维

Think Java

How to Think Like a Computer Scientist

[美] Allen B. Downey　Chris Mayfield 著
袁国忠 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

Java编程思维 / (美) 艾伦·唐尼
(Allen B. Downey) , (美) 克里斯·梅菲尔德
(Chris Mayfield) 著 ; 袁国忠译. -- 北京 : 人民邮电
出版社, 2017.1
（图灵程序设计丛书）
ISBN 978-7-115-44015-0

Ⅰ. ①J… Ⅱ. ①艾… ②克… ③袁… Ⅲ. ①JAVA语
言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2016)第276641号

内 容 提 要

本书从最基本的编程术语入手，用代码示例诠释计算机科学概念，旨在教会读者像计算机科学家那样思考，并掌握解决问题这一重要技能。书中内容共分为14章、3个附录，每章末都附有术语表和练习。

本书适合想学习计算机科学和编程相关内容的初学者。

◆ 著 [美] Allen B. Downey Chris Mayfield
译 袁国忠
责任编辑 朱 巍
执行编辑 杨 婷
责任印制 彭志环

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：13.75
字数：325千字 2017年 1 月第 1 版
印数：1－4 000册 2017年 1 月北京第 1 次印刷

著作权合同登记号 图字：01-2016-8300号

定价：59.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

本书是针对初学者编写的计算机科学和编程入门教程。从最基本的概念入手，每个术语都在首次使用时给出详尽的定义；循序渐进地介绍新概念；将内容广泛的主题（如递归和面向对象编程）分成多个部分，并分多章介绍。

本书简明扼要，每章都只有十几页的篇幅，涵盖了一周的大学课程内容。本书无意全面介绍 Java，只是想让读者了解基本的编程结构和技巧。我们从小问题和基本算法着手，逐步过渡到面向对象设计，用计算机教学术语讲，本书采取的是“迟来的对象”法。

编写理念

本书是基于如下的指导原则编写的。

- *每次一个概念*。对于可能给初学者带来麻烦的主题，将其分成多个部分，让读者无需熟悉整个主题就能将新学到的概念付诸实践。
- *兼顾 Java 和概念*。本书的主要目的并非介绍 Java，而是用代码示例诠释计算机科学概念。大多数章节以 Java 的语言特性开头，以概念结束。
- *简明扼要*。本书的一个重要目标是使篇幅够小，好让读者一个学期就能读完并搞懂本书内容。
- *突出术语*。尽可能少引入术语，并在首次使用时给出术语的详尽定义。在每章末尾，我们还将它们组织成了术语表。
- *程序开发策略*。程序编写策略有很多，包括自下而上、自上而下，等等。我们演示了开发程序的多种方法，让读者能够从中选择最适合的。
- *多条学习曲线*。要编写程序，得理解算法、熟悉编程语言，还要能够调试代码。本书始终在讨论这些内容，同时专辟了一个附录来总结调试建议。

面向对象编程

有些 Java 书一上来就介绍类和对象，有些则先介绍过程性编程，再逐步过渡到面向对象编程。

Java 的很多面向对象功能都旨在解决以前的语言存在的问题，因此，其实现受到了这些历史原因的影响。对于这些功能，如果你不熟悉它们所能解决的问题，就很难理解。

我们每次介绍一个概念，并尽可能将它讲清楚，让读者能够立即将学到的知识付诸实践。在这个前提之下，我们会尽早地介绍面向对象编程，因此，你不可能翻开本书就接触到这个主题。

然而，如果不使用面向对象功能，根本就无法编写 Java 程序，哪怕是简单的 Hello World 程序。对于有些功能，我们会在首次提及时简要地介绍一下，再在后面作更深入的讨论。

本书几乎涵盖了“AP Java subset”中的每个主题，非常适合用来备考 AP 计算机科学 A 考试（包括面向对象设计和实现）。我们的网站 http://thinkjava.org 中列出了本书各节与 AP 课程最新描述的对应关系。

附录

本书适合按顺序逐章阅读，因为每一章都以前一章的内容为基础。本书还有三个附录，你可在任何时间阅读。

- 附录 A（开发工具）

  编译、运行和调试 Java 代码的步骤随开发环境和操作系统而异，我们没有将这些细节放在正文中，因为这会分散读者的注意力。相反，我们专辟了附录 A，简要地介绍 DrJava——一个非常适合初学者使用的集成开发环境（interactive development environment，IDE），以及用于检查代码质量的 Checkstyle 和用于测试的 JUnit 等工具。

- 附录 B（Java 2D 图形）

  Java 提供了处理图形和动画的库，这些主题可能对学生很有吸引力。这些库涉及面向对象功能，读者可能阅读完前 11 章才能完全理解，但可以很早地使用它们。

- 附录 C（调试）

  有关调试的建议遍布全书，我们将这些调试建议收集到了附录 C 中。建议读者在阅读本书的过程中反复温习该附录。

使用代码示例

本书的示例代码大都可在 Git 仓库 https://github.com/AllenDowney/ThinkJavaCode 中找到。Git 是一个版本控制系统，让你能够跟踪项目中的文件。受 Git 控制的文件集合称为“仓库”。

GitHub 是一种托管服务，为 Git 仓库提供存储空间，还提供了方便的 Web 界面。它提供了多种处理代码的方式。

- 单击 Fork 键可以在 GitHub 上创建仓库的副本。如果你没有 GitHub 账户，就需要创建一个。建立分支后，你便在 GitHub 上有自己的仓库了，可用它来跟踪你编写的代码。然后，还可以“克隆”这个仓库，即将文件的副本下载到计算机。
- 你也可以在不建立分支的情况下克隆仓库。这样就不需要 GitHub 账户了，但也无法将所做的修改保存到 GitHub 中。
- 如果你根本不想使用 Git，可用 GitHub 页面上的 Download ZIP 按钮下载 ZIP 格式的代码，也可通过链接 http://www.tinyurl.com/ThinkJavaCodeZip 下载。

克隆仓库或解压 ZIP 文件后，你将看到一个名为 ThinkJavaCode 的目录，其中包含与本书每章对应的子目录。

本书中的所有示例都是用 Java SE Development Kit 8 开发和测试的。如果你使用的是更新的版本，这些示例也能正确地运行；但如果你使用的是更早的版本，有些示例可能无法正确地运行。

排版约定

本书使用了下列排版约定。

- *楷体*
  表示术语或重点强调的内容。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将Safari Books Online视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解Safari Books Online的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920041610.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多O'Reilly图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

很多人提出了勘误和建议，这里要特别感谢他们的宝贵反馈！

- Ellen Hildreth 在威尔斯利女子学院讲授数据结构时，曾将本书作为补充读物，她指出了一大堆错误，还提出了一些很不错的建议。
- Tania Passfield 指出有些术语表包含了正文中没有出现的术语。
- Elizabeth Wiethoff 注意到 $\exp(-x^2)$ 的级数展开不对，并审阅了本书的 Ruby 版。
- Matt Crawford 发来了一个包含大量勘误的文件。
- Chi-Yu Li 指出了一个代码示例的输入错误和编程错误。
- Doan Thanh Nam 更正了一个示例。
- Muhammad Saied 在将本书翻译成阿拉伯语的过程中发现了多个错误。
- Marius Margowski 发现有个代码示例存在不一致的问题。
- Leslie Klein 发现了 $\exp(-x^2)$ 的级数展开中的另一个错误，指出了 Card 数组示意图中的输入错误，并就如何让几个练习更清晰提出了很有帮助的建议。
- Micah Lindstrom 指出了六七个输入错误，并给出了更正建议。
- James Riely 将本书电子版从 LaTeX 格式转换成了 Sphinx 格式：http://fpl.cs.depaul.edu/jriely/thinkapjava/。
- Peter Knaggs 将本书转换成了 C# 版：http://www.rigwit.co.uk/think/sharp/。
- Heidi Gentry-Kolen 拍摄了多个以本书为教材的教学视频：https://www.youtube.com/user/digipipeline。

这里要特别感谢技术审阅人 Blythe Samuels、David Wisneski 和 Stephen Rose。他们找出了错误，提出了很多宝贵建议，让本书的质量得到了极大的提高。

另外，感谢以下人员发现并指出了输入错误：Stijn Debrouwere、Guy Driesen、Andai Velican、Chris Kuszmaul、Daniel Kurikesu、Josh Donath、Rens Findhammer、Elisa Abedrapo、Yousef BaAfif、Bruce Hill、Matt Underwood、Isaac Sultan、Dan Rice、Robert Beard 和 Daniel Pierce。

如果你有其他建议或看法，请发送到 feedback@greenteapress.com。

电子书

扫描如下二维码，即可购买本书电子书。



第 1 章

编程之道

本书旨在教你像计算机科学家那样思考。这种思维方式兼具数学、工程和自然科学的优点：计算机科学家像数学家那样使用规范的语言来描绘概念，具体地说就是计算；像工程师那样设计，将各个部分组装成系统并权衡不同的解决方案；像科学家那样观察复杂系统的行为，进而作出假设并进行验证。

对计算机科学家来说，最重要的技能是*解决问题*（problem solving）。这包括系统地阐述问题、创造性地提出解决方案，以及清晰而准确地描述解决方案。实践表明，学习编程为获得解决问题的技能提供了极佳的机会，这正是本章名为“编程之道”的原因所在。

一方面，你将学习编程，这本身就是一项很有用的技能；另一方面，你将把编程作为达到目的的手段。随着不断往下阅读，目的将变得更加清晰。

1.1 何为编程

程序（program）由一系列指令组成，指定了如何执行计算。这里的计算可能是数学计算，如求解方程组或找出多项式的根，也可能是符号计算，如在文档中搜索并替换文本或编译程序（真够奇怪的，编译程序竟然也是计算）。虽然细节因语言而异，但几乎所有语言都支持一些基本指令。

- *输入*

  从键盘、文件、传感器或其他设备获取数据。

- *输出*

  在屏幕上显示数据，或者将数据发送给文件或其他设备。

- *数学运算*

  执行基本的数学运算，如加法和除法。

- *决策*

  检查特定的条件，并根据检查结果执行相应的代码。

- *重复*

  反复执行某种操作，但通常每次执行时都略有不同。

信不信由你，这几乎就是程序的全部内容。你使用的每个程序都由类似于上面的小指令组成，不管它有多复杂。因此，你可将*编程*（programming）视为这样的过程，即将复杂而庞大的任务分解为较小的子任务。不断重复这个过程，直到分解得到的子任务足够简单，用计算机提供的基本指令就能完成。

1.2 何为计算机科学

对编程而言，最有趣的一个方面是决定如何解决特定的问题，尤其是问题存在多种解决方案时。例如，对数字列表进行排序的方法很多，其中每种方法都有其优点。要确定哪种方法是特定情况下的最佳方法，你必须具备规范地描述和分析解决方案的技能。

计算机科学（computer science）就是算法科学，包括找出算法并对其进行分析。*算法*（algorithm）由一系列指定如何解决问题的步骤组成。有些算法的速度比其他算法快，有些使用的计算机内存更少。面对以前没有解决过的问题，你在学着找出算法的同时，也将学习如何像计算机科学家那样思考。

设计算法并编写代码很难，也很容易出错。由于历史的原因，编程错误被称为 bug，而找出并消除编程错误的过程被称为*调试*（debugging）。通过学习调试程序，你将获得解决新问题的技能。面临出乎意料的错误时，需要创造性思维。

虽然调试可能令人沮丧，但它是计算机编程中有趣且挑战智商的部分。从某种程度上来说，调试犹如侦破工作：必须根据掌握的线索猜想出引发结果的过程和事件。在有些情况下，在考虑如何修复程序以及改善其性能的过程中，还能发现新的算法。

1.3 编程语言

本书要介绍的编程语言是 Java，这是一种*高级语言*（high-level language）。你可能还听说过其他高级语言，如 Python、C、C++、Ruby 和 JavaScript。

要想运行用高级语言编写的程序，必须将其转换为*低级语言*（low-level language），即“机器语言”。这种转换需要一定的时间，这是高级语言的一个小小的缺点，但高级语言有两个优点。

- 用高级语言编程容易得多：编写程序所需要的时间更短，程序更简洁、更容易理解，同时更容易确保程序正确无误。
- 高级语言是*可移植的*（portable），这意味着用高级语言编写的程序只需做少量修改甚至无需修改，就可在不同类型的计算机上运行。用低级语言编写的程序只能在一种计算机上运行，这种程序必须重写才能在其他计算机上运行。

有两种将高级语言转换为低级语言的程序：解释器和编译器。*解释器*（interpreter）读取并执行用高级语言编写的程序，这意味着程序怎么说它就怎么做。它每次处理程序的一小部分，即交替地读取代码行并执行计算。图 1-1 展示了解释器的结构。

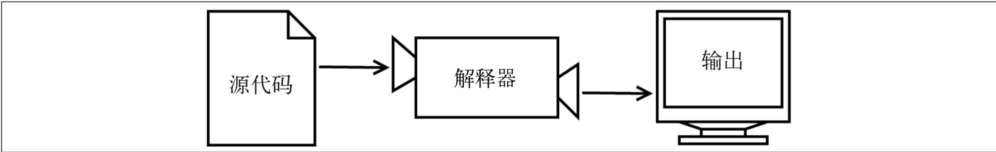


图 1-1：解释型语言是如何执行的

相反，*编译器*（compiler）读取并转换整个程序，然后才开始运行程序。在这种情况下，用高级语言编写的程序称为*源代码*（source code），而转换得到的程序称为*目标代码*（object code）或*可执行程序*（executable）。程序编译后可反复执行，无需在每次执行前都进行转换。因此，编译型程序的运行速度通常比解释型程序更快。

Java 既是解释型的又是编译型的。Java 编译器不将程序直接转换为机器语言，而是生成*字节码*（byte code）。字节码类似于机器语言，解释起来既轻松又快捷，同时也是可移植的，即可在一台机器上编译程序，在另一台机器上运行生成的字节码。运行字节码的解释器被称为 Java 虚拟机（Java Virtual Machine，JVM）。

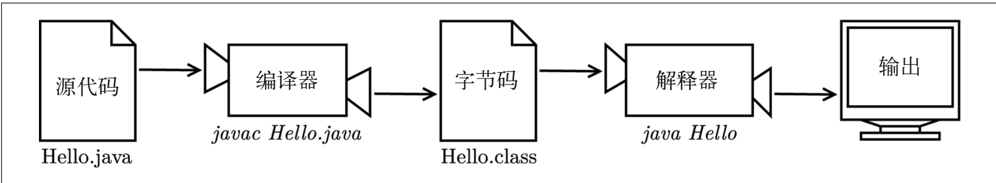


图 1-2：Java 程序的编译和运行过程

图 1-2 展示了这个过程包含的步骤。这个过程看似复杂，但在大多数程序开发环境中，这些步骤都是自动完成的：通常只需按一下按钮或输入简单的命令，就能编译并运行程序。然而，知道幕后执行了哪些步骤很重要，这样就可以在出现问题时找出问题所在。

1.4　Hello World程序

传统上，学习一门新的编程语言时，通常先编写一个名为 Hello World 的程序，它所做的只是在屏幕上显示“Hello, World!”。用 Java 编写时，这个程序与下面的类似：

```
public class Hello {

    public static void main(String[] args) {
        // 生成一些简单的输出
        System.out.println("Hello, World!");
    }
}
```

这个程序运行时显示如下内容：

```
Hello, World!
```

注意，输出中没有引号。

Java 程序由*类定义*和*方法定义*组成，而其中的方法由*语句*（statement）组成。*语句*是一行执行基本操作的代码。在 Hello World 程序中，这是一条*打印语句*（print statement），在屏幕上显示一条消息：

```
System.out.println("Hello, World!");
```

`System.out.println` 在屏幕上显示结果，其中的 `println` 表示“打印一行”。令人迷惑的是，*打印*既可以表示“在屏幕上显示”，也可以表示“发送到打印机”。在本书中，表示输出到屏幕上时，我们尽可能说“显示”。与大多数语句一样，打印语句也以分号（;）结尾。

Java 是区分大小写的，这意味着大写和小写是不同的。在前面的示例中，`System` 的首字母必须大写，使用 `system` 或 `SYSTEM` 都行不通。

方法（method）是一系列命名的语句。前面的程序定义了一个名为 `main` 的方法：

```
public static void main(String[] args)
```

方法 `main` 比较特殊：程序运行时，首先执行方法 `main` 中的第一条语句，并在执行完这个方法的最后一条语句后结束。在本书的后文中，你将看到定义了多个方法的程序。

类（class）是方法的集合。前面的程序定义了一个名为 `Hello` 的类。你可以随便给类命名，但根据约定，类名的首字母应大写。类必须与其所属的文件同名，因此前面的类必须存储在文件 Hello.java 中。

Java 用大括号（{ 和 }）编组。在 Hello.java 中，外面的大括号包含类定义，而里面的大括号包含方法定义。

以双斜线（//）开头的行是*注释*（comment），它用自然语言编写的文本解释代码。编译器遇到 // 时，将忽略随后到行尾的所有内容。注释对程序的执行没有任何影响，但可以让其他程序员（还有未来的你自己）更容易地明白你要做什么。

1.5 显示字符串

方法 `main` 中可包含任意条语句。例如，要显示多行输出，你可以像下面这样做：

```
public class Hello {

    public static void main(String[] args) {
        // 生成一些简单的输出
        System.out.println("Hello, World!"); // 第一行
        System.out.println("How are you?");  // 第二行
    }
}
```

这个示例表明，除独占一行的注释外，还可在行尾添加注释。

用引号括起的内容称为*字符串*（string），因为它们包含一系列串在一起的字符。字符包括字母、数字、标点、符号、空格、制表符，等等。

`System.out.println` 在指定的字符串末尾添加了一个特殊字符——*换行符*（newline），其作用是移到下一行开头。如果你不想在末尾添加换行符，可用 `print` 代替 `println`：

```
public class Goodbye {

    public static void main(String[] args) {
        System.out.print("Goodbye, ");
        System.out.println("cruel world");
    }
}
```

在这个示例中，第一条语句没有添加换行符，因此输出只有一行：`Goodbye, cruel world`。请注意，第一个字符串末尾有一个空格，这也包含在输出中。

1.6 转义序列

可用一行代码显示多行输出，只需告诉 Java 在哪里换行就可以了：

```
public class Hello {

    public static void main(String[] args) {
        System.out.print("Hello!\nHow are you doing?\n");
    }
}
```

上述代码的输出为两行，每行都以换行符结尾：

```
Hello!
How are you doing?
```

`\n` 是一个*转义序列*（escape sequence），表示特殊字符的字符序列。反斜线让你能够对字符串的内容进行转义。请注意，`\n` 和 `How` 之间没有空格；如果在这里添加一个空格，第二行输出的开头将会是一个空格。

转义序列的另一个常见用途是在字符串中包含引号。由于双引号标识字符串的开头和结尾，因此，要想在字符串中包含双引号，必须用反斜线对其进行转义。

```
System.out.println("She said \"Hello!\" to me.");
```

结果如下：

```
She said "Hello!" to me.
```

表1-1：常见的转义序列

`\n`	换行符
`\t`	制表符
`\"`	双引号
`\\`	反斜线

1.7　设置代码格式

在 Java 程序中，有些空格是必不可少的。例如，不同的单词之间至少得有一个空格，因此下面的程序是不合法的：

```
publicclassGoodbye{

    publicstaticvoidmain(String[] args) {
        System.out.print("Goodbye, ");
        System.out.println("cruel world");
    }
}
```

但其他空格大都是可有可无的。例如，下面的程序完全合法：

```
public class Goodbye {
public static void main(String[] args) {
System.out.print("Goodbye, ");
System.out.println("cruel world");
}
}
```

换行也是可选的，因此可将前面的代码编写如下：

```
public class Goodbye { public static void main(String[] args)
{ System.out.print("Goodbye, "); System.out.println
("cruel world");}}
```

这也是可行的，但程序阅读起来更难了。要以直观的方式组织程序，换行和空格都很重要，使用它们可让程序更容易理解，发生错误时也更容易查找。

很多编辑器都自动设置源代码的格式：以一致的方式缩进和换行。例如，在 DrJava（参见附录 A）中，可以按 Ctrl+A 选择所有的代码，再按 Tab 键来缩进代码。

从事大量软件开发工作的组织通常会制定严格的源代码格式设置指南，例如，Google 就发布了针对开源项目的 Java 编码标准，其网址为 http://google.github.io/styleguide/javaguide.html。

这些指南提及了本书还未介绍的 Java 功能，因此你现在可能看不懂，但在阅读本书的过程中，你可能时不时地想要回过头来阅读它们。

1.8 调试代码

最好能在计算机前阅读本书，因为这样你就可以一边阅读一边尝试其中的示例。本书中的很多示例可直接在 DrJava 的 Interactions 窗格（见附录 A）中运行，但如果将代码存储到源代码文件中，则更容易对其修改再运行。

每当你使用新功能时，都应该尝试故意犯些错误。例如，在 Hello World 程序中，如果遗漏一个引号，结果将如何呢？如果两个引号都遗漏了，结果将如何呢？如果 `println` 拼写得不正确，结果又将如何呢？这些尝试不仅有助于牢记学过的知识，还有助于调试程序，因为你将知道各种错误消息意味着什么。现在故意犯错胜过以后无意间犯错。

调试犹如实验科学：一旦对出问题的地方有所感觉，就修改程序并再次运行。如果假设没错，你就能预测修改后的结果，从而离程序正确运行更近一步；如果假设有误，你就必须作出新的假设。

编程和调试必须齐头并进。不能先随便编写大量的代码，再通过反复调试来确保它们能够正确地运行；相反，应先编写少量可正确运行的代码，再逐步修改和调试，最终得到一个提供所需功能的程序。这样的方式可以确保在任何时候都有可运行的程序，从而更容易隔离错误。

Linux 操作系统淋漓尽致地展示了这种原则。这个操作系统现在包含数百万行的代码，但最初只是一个简单的程序，Linus Torvalds 用它来研究 Intel 80386 芯片。正如 Larry Greenfield 在 *Linxu User's Guide* 中指出的，Linux 是 Linus Torvalds 早期开发的项目之一，最初只是一个决定打印 AAAA 还是 BBBB 的程序，后来才演变为 Linux。

最后，编程可能引发强烈的情绪。面对棘手的 bug 而束手无策时，你可能会感到愤怒、沮丧或窘迫。别忘了，并非只有你这样，大多数乃至所有程序员都有类似的经历；不要犹豫，赶快向朋友求助吧！

1.9 术语表

对于每个术语，本书都尽可能在首次用到时作出定义。同时，我们会在每章末尾按出现顺序列出涉及的新术语及其定义。如果你花点时间研究以下术语表，后面的内容阅读起来将更加轻松。

- 解决问题
  明确地描述问题、找到并描述解决方案的过程。
- 程序
  一系列的指令，指定了如何在计算机上执行任务。
- 编程
  用问题解决技能创建可执行的计算机程序。
- 计算机科学
  科学而实用的计算方法及其应用。
- 算法
  解决问题的流程或公式，可以涉及计算机，也可以不涉及。
- bug
  程序中的错误。
- 调试
  找出并消除错误的过程。
- 高级语言
  人类能够轻松读写的编程语言。
- 低级语言
  计算机能够轻松运行的编程语言，也叫“机器语言”或“汇编语言”。
- 可移植
  程序能够在多种计算机上运行。
- 解释
  指运行用高级语言编写的程序，即每次转换其中的一行并立即执行转换得到的指令。

- *编译*

  将用高级语言编写的程序一次性转换为低级语言，供以后执行。

- *源代码*

  用高级语言编写的、未编译的程序。

- *目标代码*

  编译器转换程序后得到的输出。

- *可执行代码*

  可在特定硬件上执行的目标代码的别名。

- *字节码*

  Java 程序使用的一种特殊目标代码，类似于低级语言，但像高级语言一样是可移植的。

- *语句*

  程序的一部分，指定了算法中的一个步骤。

- *打印语句*

  将输出显示到屏幕上的语句。

- *方法*

  命名的语句序列。

- *类*

  就目前而言，指的是一系列相关的方法。（后面你将看到，类并非只包含方法。）

- *注释*

  程序的一部分，包含有关程序的信息，但对程序的运行没有任何影响。

- *字符串*

  一系列字符，是一种基本的文本数据类型。

- *换行符*

  标识文本行尾的特殊字符。

- *转义序列*

  在字符串中用于表示特殊字母的编码序列。

1.10 练习

每章末尾都有练习，只需要利用在该章学到的知识就能完成。强烈建议你尝试完成每个练

习，光阅读是学不会编程的，得实践才行。

要想编译并运行 Java 程序，需要下载并安装一些工具。这样的工具很多，但我们推荐 DrJava——一个非常适合初学者使用的“集成开发环境”。有关如何安装 DrJava，请参阅附录 A 的 A.1 节。

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch01 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习1-1

计算机科学家有个毛病，喜欢赋予常见的英语单词以新的义项。例如，在英语中 statement 和 comment 是同义词，但在程序中它们的含义不同。

(1) 在计算机行话中，语句和注释有何不同？
(2) 说程序是可移植的是什么意思？
(3) 在普通英语中，单词 compile 是什么意思？
(4) 何为可执行程序（executable）？为何这个单词被用作名词？

每章末尾的术语表旨在突出计算机科学中有特殊含义的单词和短语。看到熟悉的单词时，千万不要理所应当地认为你知道它们的含义！

练习1-2

接着往下读之前，请先搞清楚如何编译和运行 Java 程序。有些编程环境提供了类似于本章 Hello World 程序的示例程序。

(1) 输入 Hello World 程序的代码，再编辑并运行它。
(2) 添加一条打印语句，在“Hello, World!”后面再显示一条诙谐的消息，如“How are you?”，然后再编译并运行这个程序。
(3) 在这个程序中添加一条注释（什么地方都可以），再编译并运行它。新添的注释应该对结果没有任何影响。

这个练习看似微不足道，却为编写程序打下了坚实的基础。要想得心应手地调试程序，必须熟悉编程环境。

在一些编程环境中，一不小心就不知道当前执行的是哪个程序了。你可能想调试某个程序，却不小心运行了另一个程序。为确保你看到的就是要运行的程序，一种简单的方法是添加并修改打印语句。

练习1-3

将能想到的错误都犯一次是个不错的注意，这样你就知道编译器都会显示哪些错误消息了。在有些情况下，编译器会准确地指出错误，你只需要修复指出的错误即可；但有时

候，错误消息会将你引入歧途。调试多了就会培养出感觉，知道什么情况下该信任编译器，什么情况下只能自力更生。

请在本章的 Hello World 程序中尝试下面每一种错误。每次修改后编译程序并阅读出现的错误消息，然后再修复错误。

(1) 删除其中的一个左大括号。
(2) 删除其中的一个右大括号。
(3) 将方法名 `main` 改为 `mian`。
(4) 删除单词 `static`。
(5) 删除单词 `public`。
(6) 删除单词 `System`。
(7) 将 `println` 改为 `Println`。
(8) 将 `println` 替换为 `print`。
(9) 删除其中的一个括号；添加一个括号。

第2章

变量和运算符

本章将介绍如何用变量和运算符来编写语句。变量用于存储数字、单词等值，而运算符是执行计算的符号。另外，本章还将介绍三种编程错误，并提供其他的调试建议。

2.1 声明变量

编程语言最强大的功能之一是能够定义和操作*变量*（variable）。变量是存储值（value）的命名位置，其中的值可以是数字、文本、图像、声音和其他类型的数据。要存储值，得先声明变量。

```
String message;
```

这条语句是一个*声明*（declaration），因为它声明变量 `message` 的类型为 `String`。每个变量都有*类型*（type），决定了它可以存储什么样的值。例如，类型为 `int` 的变量可存储整数，而类型为 `char` 的变量可存储字符。

有些类型名的首字母大写，有些类型名的首字母小写。这种差别的含义将在后文中介绍，就目前而言，你只需要确保首字母大小写正确即可，因为没有类型 `Int`，也没有类型 `string`。

要声明整型变量，可用如下语法：

```
int x;
```

其中的 `x` 是一个随便指定的变量名。一般而言，使用的名称应指出变量的含义。例如，看

到下面的声明，你可能就能猜出各个变量将存储什么值：

```
String firstName;
String lastName;
int hour, minute;
```

这个示例声明了四个变量，其中两个的类型为 String，另外两个的类型为 int。根据约定，对于包含多个单词的变量名，如 firstName，应将每个单词的首字母大写，但第一个单词除外。变量名是区分大小写的，因此，firstName、firstname 和 FirstName 指的是不同的变量。

这个示例还演示了在一行中声明多个同类变量的语法：hour 和 minute 都是 int 变量。请注意，每条声明语句都以分号结尾。

你可以随便给变量命名，但大约有 50 个被称为*关键词*（keyword）的保留词不能用作变量名。这些关键词包括 public、class、static、void 和 int，　被编译器用来分析程序的结构。

有关完整的关键词清单，请参阅 http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html，但不必记住它们。大多数编程编辑器都提供了“语法突出”的功能，即用不同的颜色显示程序的不同部分。

2.2 赋值

声明变量后，即可用它们来存储值。为此，可用*赋值*（assignment）语句：

```
message = "Hello!"; // 给变量message指定值"Hello!"
hour = 11;          // 将值11赋给变量hour
minute = 59;        // 将变量minute的值设置为59
```

这个示例包含三条赋值语句，其中的注释指出了三种解读赋值语句的方式。这里使用的术语可能令人迷惑，但涉及的概念简单易懂。

- 当声明变量时，便创建了一个命名的存储位置。
- 当给变量赋值时，便修改了它的值。

一般而言，变量和赋给它的值必须是相同的类型。例如，你不能将字符串存储到变量 mintue 中，也不能将整数存储到变量 message 中。在本书的后文中，你将看到一些违反这条规则的示例。

有些字符串*看起来像*是整数，但其实不是整数，这常常令人迷惑。例如，变量 message 可包含字符串 "123"，这个字符串由字符 '1'、'2' 和 '3' 组成，与整数 123 不是同一码事。

```
message = "123"; // 合法
message = 123;   // 非法
```

使用变量前，必须对其进行*初始化*（initialize，首次赋值）。你可以像前一个示例那样，先声明变量，再赋值；也可以在声明变量的同时给它赋值：

```
String message = "Hello!";
int hour = 11;
int minute = 59;
```

2.3　状态图

鉴于 Java 用符号“=”来赋值，你可能会认为语句 `a=b` 是一个相等声明，但事实并非如此！

相等具有交换性，但赋值并非如此。例如，在数学中，如果 $a=7$，则 $7=a$；而在 Java 中，`a=7;` 是一条合法的赋值语句，但 `7=a;` 则不是，因为赋值语句的左边必须是变量名（存储位置）。

另外，在数学中，相等声明在任何情况下都成立。如果当前 $a=b$，那么在任何情况下 a 和 b 都相等；而在 Java 中，赋值语句可能导致两个变量相等，但它们并不一定始终如此。

```
int a = 5;
int b = a;      // 现在a和b相等
a = 3;          // a和b不再相等
```

第三行代码修改了 `a` 的值，但没有修改 `b` 的值，因此它们不再相等。

程序中的所有变量及其当前值一同组成了程序的*状态*（state）。图 2-1 显示了程序在上述三条语句运行后的状态。

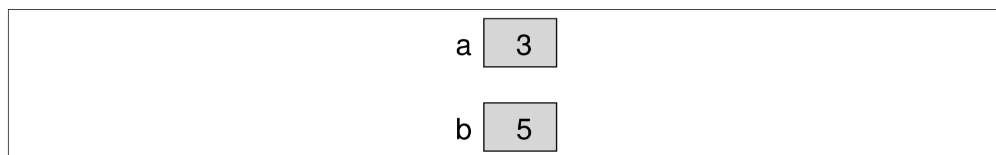


图 2-1：变量 `a` 和 `b` 的状态图

显示程序状态的图被称为*状态图*（state diagram）。每个变量都用一个方框表示，方框内是变量的值，方框外是变量名。状态随程序的运行而变化，因此，应将状态图视为程序执行过程中特定时点的快照。

2.4　显示变量

可用 `print` 或 `println` 显示变量的值。下面的语句声明了一个名为 `firstLine` 的变量，将值 `"Hello, again!"` 赋给它，并显示这个值：

```
String firstLine = "Hello, again!";
System.out.println(firstLine);
```

在说显示变量时，我们通常指的是显示变量的*值*。要显示变量的*名称*，必须将其用引号括起。

```
System.out.print("The value of firstLine is ");
System.out.println(firstLine);
```

这个示例的输出如下：

```
The value of firstLine is Hello, again!
```

不管变量的类型如何，显示其值的语法都相同。例如：

```
int hour = 11;
int minute = 59;
System.out.print("The current time is ");
System.out.print(hour);
System.out.print(":");
System.out.print(minute);
System.out.println(".");
```

这个程序的输出如下：

```
The current time is 11:59.
```

要在同一行输出多个值，通常使用多条 `print` 语句，并在最后使用一条 `println` 语句。千万不要忘了 `println` 语句！很多计算机都将来自 `print` 的输出存储起来，等遇到 `println` 后才将整行输出一次性显示出来。如果省略了 `println`，程序可能在意想不到的时候显示存储的输出，甚至直到结束也不显示任何输出。

2.5　算术运算符

运算符（operator）是表示简单计算的符号，例如，加法运算符为 `+`，减法运算符为 `-`，乘法运算符为 `*`，而除法运算符为 `/`。

下面的程序将时间转换为分钟数：

```
int hour = 11;
int minute = 59;
System.out.print("Number of minutes since midnight: ");
System.out.println(hour * 60 + minute);
```

在这个程序中，`hour * 60 + minute` 是一个*表达式*（expression），表示计算将得到的单个值。这个程序运行时，每个变量都被替换为当前值，再执行运算符指定的计算。运算符使用的值称为*操作数*（operand）。

前述示例的结果如下：

```
Number of minutes since midnight: 719
```

表达式通常由数字、变量和运算符组成。程序编译并执行时，表达式将变成单个值。

例如，表达式 `1 + 1` 的值为 `2`。对于表达式 `hour - 1`，Java 将变量 `hour` 替换为其当前值，因此这个表达式变成 `11 - 1`，结果为 `10`。对于表达式 `hour * 60 + minute`，其中的两个变量都被替换了，整个表达式变为 `11 * 60 + 59`。先执行乘法运算，因此这个表达式变为 `660 + 59`；再执行加法运算，结果为 `719`。

加法、减法、乘法运算都与你的预期一样，但除法运算可能会让你感到意外。例如，下面的代码片段试图将分钟数转换为小时数：

```
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute / 60);
```

其输出如下：

```
Fraction of the hour that has passed: 0
```

这样的结果令人感到迷惑。变量 `minute` 的值为 59，59 除以 60 的结果应为 0.98333，而不是 0。问题在于 Java 在两个操作数都为整数时执行“整数除法”，而根据设计，整数除法总是向下圆整，即便这里的结果更接近下一个整数时也是如此。

一种替代方式是，计算百分比而不是小数：

```
System.out.print("Percent of the hour that has passed: ");
System.out.println(minute * 100 / 60);
```

上述代码的输出如下：

```
Percent of the hour that has passed: 98
```

同样，结果也被向下圆整了，但至少离正确的答案更近了。

2.6 浮点数

一种更通用的解决方案是使用**浮点**（floating-point）数，它可用于表示小数，也可用于表示整数。在 Java 中，默认的浮点类型为 `double`，它指的是双精度浮点数。`double` 变量的声明和赋值语法与其他类型的变量相同：

```
double pi;
pi = 3.14159;
```

只要有一个操作数为 double 值，Java 就执行“浮点除法”，因此，我们可以用如下方式来解决 2.5 节中的问题：

```
double minute = 59.0;
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute / 60.0);
```

输出如下：

```
Fraction of the hour that has passed: 0.9833333333333333
```

虽然浮点数很有用，但也可能让人感到迷惑。例如，Java 区分整数 1 和浮点数 1.0，即使它们看起来是同一个数字。它们属于不同的数据类型，而严格来说，你不能将一种类型的值赋给另一种类型的变量。

下面的语句是非法的，因为左边是一个 int 变量，而右边是一个 double 值：

```
int x = 1.1; // 编译错误
```

这种规则很容易忘记，因为 Java 在很多情况下会自动转换类型：

```
double y = 1; // 合法,但这是一种糟糕的做法
```

这个示例原本应该是非法的，但由于 Java 自动将 int 值 1 转换为 double 值 1.0，使得这个示例变得合法了。这样的宽容是十分便利的，但常会给初学者带来问题，例如：

```
double y = 1 / 3; // 常见的错误
```

你可能会认为变量 y 的值为 0.333333——一个合法的浮点值，但实际上其值为 0。右边的表达式将两个整数相除，因此 Java 执行整数除法，结果为 int 值 0。这个结果被转换为 double 值 0.0，再赋给变量 y。

对于这种问题，其中一种解决方案是将右边的表达式变成浮点表达式，如下所示，这样变量 y 将像预期的那样被设置为 0.333333：

```
double y = 1.0 / 3.0; // 正确
```

作为一种编程风格，在任何情况下都应将浮点值赋给浮点变量。编译器并没有要求必须这样做，但如果不这样做的话，不知什么时候一个简单的错误就可能阴魂不散，给你带来麻烦。

2.7 舍入误差

大多数浮点数只能*大致*正确地表示。有些数字，如果不是特别大的整数，可以准确地表示。但循环小数（如 1/3）和无理数（如 π）不能准确地表示。为表示这些数字，计算机必

须将其舍入到最接近的浮点数。

所需数字和实际得到的浮点数之间的差称为*舍入误差*（rounding error）。例如，以下两条语句应该是等价的：

```
System.out.println(0.1 * 10);
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1
                 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
```

但在很多计算机上，它们的输出如下：

```
1.0
0.9999999999999999
```

原因是 `0.1` 在十进制中为有限小数，但在二进制中为循环小数，因此其浮点数表示只是近似的。将近似值相加会逐渐累积舍入误差。

在很多应用领域，如计算机图形学、加密、统计分析和多媒体渲染，使用浮点数算术利大于弊。但如果要求绝对精确，应使用整数。例如，查看余额为 123.45 美元的银行账户：

```
double balance = 123.45; // 可能存在舍入误差
```

在这个示例中，随着不断地对变量执行算术运算（如存款和取款），它存储的值将不再准确。这可能会激怒顾客，甚至引发诉讼。为避免这种问题，可以用整数来表示余额：

```
int balance = 12345; // 美分数
```

只要美分数不超过变量 `int` 可表示的最大值（约 20 亿），就可以使用这种解决方案。

2.8 字符串运算符

一般而言，不能对字符串执行数学运算，即便对那些看起来像数字的字符串亦是如此。下面的表达式是非法的：

```
"Hello" - 1     "World" / 123    "Hello" * "World"
```

运算符 `+` 可用于字符串，但其所作所为可能出乎意料。用于字符串时，运算符 `+` 执行*串接*（concatenation），即首尾相连，因此 `"Hello, " + "World!"` 的结果为字符串 `"Hello, World!"`。

再举一个例子。如果你声明了类型为 `String` 的变量 `name`，则表达式 `"Hello, " + name` 会将变量 `name` 的值附加在字符串 hello 的后面，从而生成个性化的问候。

鉴于对数字和字符串都定义了加法运算，因此 Java 可能执行意料之外的自动转换：

```
System.out.println(1 + 2 + "Hello");
// 输出为3Hello

System.out.println("Hello" + 1 + 2);
// 输出为Hello12
```

Java 按从左到右的顺序执行这些运算。在第 1 行中，`1 + 2` 等于 `3`，而 `3 + "Hello"` 的结果为 `"3Hello"`；在第 2 行中，`"Hello" + 1` 的结果为 `"Hello1"`，而 `"Hello1" + 2` 的结果为 `"Hello12"`。

表达式包含多个运算符时，将根据*运算顺序*（order of operation）计算表达式。一般而言，Java 按从左到右的顺序执行运算（如 2.7 节所示），但对于数值运算符，Java 遵循如下的数学规则。

- 乘除运算的优先级高于加减运算，这意味着先乘除后加减。因此 `1 + 2 * 3` 的结果为 `7`，而不是 `9`，而 `2 + 4 / 2` 的结果为 `4`，而不是 `3`。
- 运算符的优先级相同时，按从左到右的顺序执行。因此，表达式 `minute * 100 / 60` 先执行乘法运算；如果 `minute` 的值为 59，这个表达式将变为 `5900 / 60`，结果为 `98`。如果按从右到左的顺序执行这些运算，将得到错误的结果 `59 * 1`。
- 要想改变默认的运算优先级或对默认的运算优先级不太确定时，可使用括号。首先计算括号内的表达式，因此 `(1 + 2) * 3` 的结果为 `9`。还可用括号让表达式更容易理解，如 `(minute * 100) / 60`，虽然就这个表达式而言，用不用括号对结果并没有影响。

别费劲地去记运算符的优先级，尤其是除算术运算符外的其他运算符。如果表达式的含义不那么明显，可用括号让它清晰起来。

2.9 组合

前面分别介绍了编程语言的一些元素——变量、表达式和语句，但没有讨论如何结合使用它们。

编程语言最有用的功能之一是能够*组合*（compose）小型构件。例如，在知道如何将数字相乘以及如何显示值后，我们可以将这些操作放在一条语句中：

```
System.out.println(17 * 3);
```

任何算术表达式都可用于打印语句中，我们见过这样的例子：

```
System.out.println(hour * 60 + minute);
```

还可将表达式放在赋值语句的右边：

```
int percentage;
percentage = (minute * 100) / 60;
```

赋值语句的左边必须是变量名，不能是表达式，这是因为赋值语句的左边要指定将结果放在什么地方，而表达式表示的并非存储位置。

```
hour = minute + 1;  // 正确
minute + 1 = hour;  // 导致编译错误
```

就目前而言，能够将操作组合起来好像没什么大不了的，但在本书的后文中你将了解到，这让我们能够编写简洁的代码以执行复杂的计算。不过，也别忘乎所以，冗长而复杂的表达式可能会难以理解和调试。

2.10 错误类型

程序中可能出现的错误有三种：编译时错误、运行时错误和逻辑错误。区分这些错误可以更快地找出错误。

编译时错误（compile-time error）指的是因违反 Java *语法*（syntax）规则而导致的错误。例如，括号和大括号必须成对出现，所以 `(1 + 2)` 是合法的，而 `8)` 是非法的。`8)` 导致程序无法编译，而编译器将显示一条错误消息。

编译器显示的错误消息通常会指出错误出现在程序的什么地方，有时还可以准确地指出错误。我们来重温一下第 1 章中的 Hello World 程序。

```
public class Hello {

    public static void main(String[] args) {
        // 生成一些简单的输出
        System.out.println("Hello, World!");
    }
}
```

如果遗漏了打印语句末尾的分号，将出现类似于以下的错误消息：

```
File: Hello.java  [line: 5]
Error: ';' expected
```

真是太好了：这条错误消息准确地指出了错误的位置，还指出了是什么样的错误。

然而，并非所有的错误消息都是容易理解的。有时编译器报告的错误位置不准确；有时对错误的描述模棱两可，几乎没什么帮助。

例如，如果遗漏了方法 `main` 末尾（第 6 行）的右大括号，可能出现类似于以下的错误消息：

```
File: Hello.java  [line: 7]
Error: reached end of file while parsing
```

这里有两个问题。首先，这条错误消息是从编译器的角度而不是你的角度生成的。分析（parsing）指的是在转换前读取程序的过程；如果编译器到达文件末尾后分析还在进行的话，那么就意味着程序遗漏了什么东西，但编译器不知道遗漏了什么，也不知道在何处遗漏的。它认为错误发生在程序末尾（第 7 行），但遗漏的大括号应该在前一行。

错误消息提供了很有用的信息，你应尽力阅读并理解它们，但也不能将它们奉为圭臬。

刚从事编程的几周内，你可能会为找出编译错误花费大量的时间，但随着经验越来越丰富，你犯的错误将越来越少，找出错误的速度也将越来越快。

第二种错误是运行时错误（run-time error），因其要等到程序运行后才会出现而得名。在 Java 中，这种错误发生在解释器执行字节码期间，也被称为异常，因为它们通常表明出现了异常而糟糕的情况。

本书前几章的简单程序中很少出现运行时错误，因此可能需要过段时间才能见到它们。运行时错误发生时，解释器将显示一条错误消息，指出在什么地方出现了什么问题。

例如，如果你不小心将零用作了除数，将出现类似于以下的错误消息：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Hello.main(Hello.java:5)
```

上述的输出对调试很有帮助。第 1 行指出了异常的名称——`java.lang.ArithmeticException`，还具体地指出了发生的情况——`/ by zero`（除以零）。接下来的一行指出了问题所在的方法；`Hello.main` 指的是 `Hello` 类的方法 `main`；还指出了这个方法是在哪个文件（Hello.java）中定义的以及问题出现在第几行（5）。

有些错误消息还包含无意义的信息，因此你面临的挑战之一是确定有用的部分，而不被多余的信息搞得不知所措。另外别忘了，导致程序崩溃的代码行可能并不是需要修改的代码行。

第三种错误是逻辑错误（logic error）。存在逻辑错误的程序能够通过编译，且运行时不会出现错误消息，但不会做正确的事。相反，你让它怎么做，它就怎么做。例如，下面这个版本的 Hello World 程序存在一个逻辑错误：

```
public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello, ");
        System.out.println("World!");
    }
}
```

这个程序能够通过编译并运行，但输出如下：

```
Hello,
World!
```

如果我们要在一行中显示全部输出，那么上述输出就不正确。问题出在第 1 行，它用的是 `println`，而我们原本想用的是 `print`（参见 1.5 节中的 goodbye world 示例）。

有时很难找出逻辑错误，因为你必须进行反向推导：根据输出结果推断程序行为不正确的原因，并确定如何让它的行为正确无误。编译器和解释器在这方面帮不了你，因为它们并不知道正确的行为是什么样的。

了解这三种错误后，你应该阅读一下附录 C，其中搜集了一些我们最喜欢的调试建议。因为这些建议涉及了一些还未讨论的语言功能，所以你可能需要时不时地再次阅读这个附录。

2.11 术语表

- 变量

  命名的存储位置。所有变量都有类型，这是在创建变量时声明的。

- 值

  数字、字符串或其他可存储在变量中的数据。每个值都属于特定的类型，如 `int` 或 `String`。

- 声明

  创建变量并指定其类型的语句。

- 类型

  从数学角度来说，类型是一个值集。变量的类型决定了它可存储哪些值。

- 关键词

  编译器用来分析程序的保留词。关键词（如 `public`、`class` 和 `void`）不能用作变量名。

- 赋值

  给变量指定值的语句。

- 初始化

  首次给变量赋值。

- 状态

  程序中的变量及其当前值。

- 状态图

  程序在特定时点的状态的图形表示。

- 运算符

  表示计算（如加、乘和字符串串接）的符号。

- 操作数

  运算符操作的值。在 Java 中，大多数运算符需要两个操作数。

- 表达式

  表示单个值的变量、运算符和值的组合。表达式也有类型，这是由表达式包含的运算符和操作数决定的。

- 浮点

  一种数据类型，表示包含整数部分和小数部分的数字。在 Java 中，默认的浮点类型为 `double`。

- 舍入误差

  要表示的数字和与之最接近的浮点数之间的差。

- 拼接

  将两个值（通常是字符串）首尾相连。

- 运算顺序

  决定运算顺序执行的规则。

- 组合

  将简单的表达式和语句合并为复合的表达式和语句。

- 语法

  程序的结构，即程序包含的单词和符号的排列方式。

- 编译时错误

  导致源代码无法编译的错误，也叫“语法错误”。

- 分析

  分析程序的结构，这是编译器做的第一项工作。

- 运行时错误

  导致程序无法完成运行的错误，也叫“异常”。

- 逻辑错误

  导致程序的行为不符合程序员预期的错误。

2.12 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch02 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

如果你还没有阅读 A.2 节，那么现在正是阅读的好时机。该节介绍了 DrJava 的 Interactions 窗格，它提供了极佳的途径，让你无需编写完整的类定义就能开发并测试简短的代码片段。

练习2-1

如果你将本书用作教材的话，可能会很喜欢这个练习。找个同伴一起来玩 Stump the Chump 的游戏吧。

先编写一个能够通过编译并正确运行的程序。一个人在程序中添加一个错误，另一个人不能偷看，然后尝试找出并修复这个错误。在不编译程序的情况下找出错误得两分，求助于编译器找出错误得 1 分，找不出错误对手得 1 分。

练习2-2

这个练习旨在：用字符串拼接显示不同类型（`int` 和 `String`）的值；以每次添加几条语句的方式循序渐进地练习程序开发。

(1) 新建一个程序，将其命名为 Date.java。输入或复制类似于程序 Hello World 中的代码，并确保程序能够通过编译并运行。
(2) 仿照 2.4 节中的示例，编写一个创建变量 `day`、`date`、`month` 和 `year` 的程序。变量 `day` 用于存储星期几（如星期五），`date` 用于存储日期（如 13 号）。这些变量应声明为何种类型呢？将表示当前日期的值赋给这些变量。
(3) 显示（打印）每个变量的值，且每个变量要独占一行。这是一个中间步骤，有助于确认到目前为止一切正确。编译并运行这个程序，然后再接着往下做。
(4) 修改程序，使其以美国标准格式显示日期，如 `Thursday, July 16, 2015`。
(5) 修改程序，使其以欧洲格式显示日期。最终的输出应类似于以下这样：

```
American format:
Thursday, July 16, 2015
European format:
Thursday 16 July 2015
```

练习2-3

这个练习旨在：使用一些算术运算符；考虑用多个值表示复合实体（如时间）。

(1) 新建一个程序，将其命名为 Time.java。从现在开始，我们将不再提醒你先编写一个可运行的小程序，但你应该这样做。
(2) 仿照 2.4 节中的示例，创建变量 `hour`、`minute` 和 `second`，并将大致表示当前时间的值赋给这些变量。请使用 24 小时制，即如果当前时间是下午两点，就将变量 `hour` 的值设置为 14。
(3) 让程序计算并显示从午夜开始过去了多少秒。
(4) 计算并显示当天还余下多少秒。
(5) 计算并显示当天已逝去时间的百分比。如果用整数计算百分比，可能会出现问题，因此请考虑使用浮点数。
(6) 根据当前时间修改变量 `hour`、`minute` 和 `second` 的值，再编写代码来计算从你开始做这个练习算起，已过去了多少时间。

提示：你可能想在计算期间用额外的变量来存储值。只用于计算而不显示的变量被称为“中间变量”或“临时变量”。

第 3 章

输入和输出

本书前面介绍的程序都只显示消息，并不涉及太多的计算。本章将介绍如何从键盘读取输入，并用这些输入来计算结果，再设置结果的输出格式。

3.1 System类

本书前面一直在使用 System.out.println，但你可能没有想过其含义。System 是一个类，提供了与运行程序的系统或环境相关的方法，以及特殊值 System.out，这个值提供了显示输出的方法，其中包括 println。

实际上，我们可用 System.out.println 来显示 System.out 的值：

```
System.out.println(System.out);
```

结果如下：

```
java.io.PrintStream@685d72cd
```

上述输出表明，System.out 是一个 PrintStream，而 PrintStream 是在 java.io 包中定义的。包（package）是一组相关的类；java.io 包含用于 I/O（输入和输出）的类。

@ 后面的数字和字母是 System.out 的十六进制地址（address）。值的地址指的是值在计算机内存中的位置，可能随计算机而异。在这个示例中，地址为 685d72cd，但如果你运行这些代码，可能会得到不同的地址。

如图 3-1 所示，System 是在文件 System.java 中定义的，而 PrintStream 是在文件

PrintStream.java 中定义的。这些文件都包含在 Java 库（library）中，Java 库是一个庞大的类集，你可以在程序中使用其中的任何类。

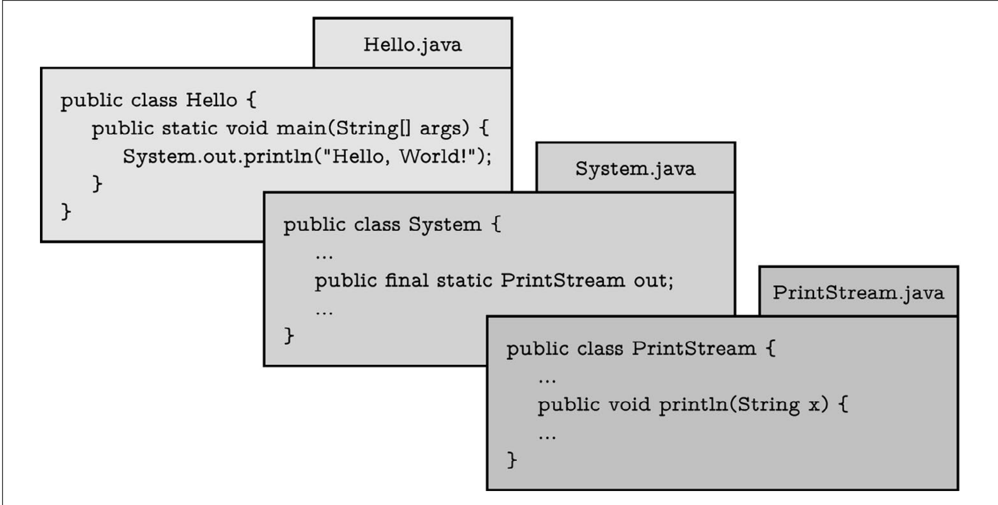


图 3-1：`System.out.println` 指向 `System` 类的 `out` 变量，这个变量是一个提供了方法 `println` 的 `PrintStream`

3.2 Scanner类

`System` 类还提供了特殊值 `System.in`，这是一个 `InputStream`，提供了从键盘读取输入的方法。这些方法用起来并不那么容易，好在 Java 还提供了其他类，从而能更容易地处理常见的输入任务。

例如，`Scanner` 类提供了输入单词、数字和其他数据的方法，其包含在 `java.util` 包中。`java.util` 包含的类很有用，因此被称为"实用类"。在使用 `Scanner` 之前，必须先像下面这样导入它：

```
import java.util.Scanner;
```

这条导入语句（import statement）告诉编译器，当说到 `Scanner` 时，你指的是 `java.util` 中定义的 `Scanner`。必须将这一点传达给编译器，因为其他包中可能也存在 `Scanner` 类。使用导入语句可避免代码存在二义性。

导入语句不能存在于类定义中。根据约定，它们通常位于文件的开头。

接下来，你需要创建一个 `Scanner` 对象：

```
Scanner in = new Scanner(System.in);
```

这行代码声明了一个名为 in 的 Scanner 变量，并新建了一个 Scanner 对象以便从 System.in 获取输入。

Scanner 提供了方法 nextLine，这个方法从键盘读取一行输入，并返回一个 String。下面的示例读取了两行，并向用户显示了它们：

```
import java.util.Scanner;

public class Echo {

    public static void main(String[] args) {
        String line;
        Scanner in = new Scanner(System.in);

        System.out.print("Type something: ");
        line = in.nextLine();
        System.out.println("You said: " + line);

        System.out.print("Type something else: ");
        line = in.nextLine();
        System.out.println("You also said: " + line);
    }
}
```

如果你在没有包含上述导入语句的情况下引用 Scanner，编译器将显示一条类似于 cannot find symbol（找不到符号）的消息，这意味着编译器不知道你说的 Scanner 指的是什么。

你可能会心存疑惑，为何不用导入 System 就能使用它呢？ System 位于可自动导入的 java.lang 包中。Java 文档指出，java.lang 提供了 Java 编程语言中的基本类。String 类也位于 java.lang 包中。

3.3 程序结构

至此，我们介绍了 Java 程序的所有组成元素，图 3-2 显示了这些组织单元。

我们来复习一下。包是类的集合，而类定义了方法；方法包含语句，而有些语句包含表达式；表达式由标记（token）组成，而标记是程序的基本元素，其中包括数字、变量名、运算符、关键词以及括号、大括号和分号等标点。

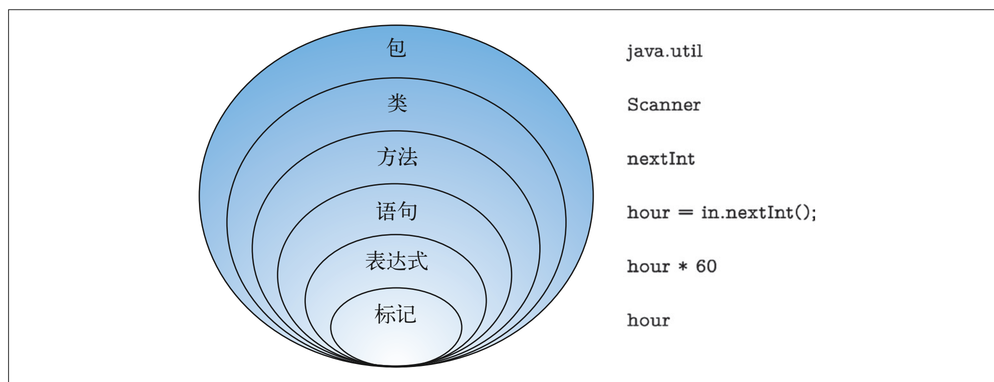


图 3-2：Java 语言的组成元素，按从大到小的顺序排列

Java 的标准版自带了可在程序中导入的数千个类，这令人既激动又惊恐。要浏览这个库，可访问 http://docs.oracle.com/javase/8/docs/api/。Java 库主要是用 Java 编写的。

请注意，Java 语言和 Java 库的主要差别在于，前者定义了图 3-2 所示元素的语法和含义，而后者提供了内置类。

3.4 英寸到厘米的转换

现在让我们来看一个有点实用价值的示例。虽然全球的大部分地区都在用公制度量衡，但有些国家依然还在用英制单位。例如，与欧洲的朋友谈论天气时，美国人可能要在摄氏温度和华氏温度之间进行转换。另外，还可能要将身高从英寸数转换为厘米数。

我们可以编写一个程序来提供帮助。在这个程序中，我们将用一个 `Scanner` 对象来获取以英寸为单位的值，然后将其转换为厘米数并显示结果。下面的代码行声明了所需要的变量并创建了一个 `Scanner` 对象：

```
int inch;
double cm;
Scanner in = new Scanner(System.in);
```

接下来需要提示用户输入值。为此，我们将使用 `print` 而不是 `println`，让用户能够在提示所在的行进行输入。另外，我们还将使用 `Scanner` 类的方法 `nextInt`，以便从键盘读取输入并将其转换为整数：

```
System.out.print("How many inches? ");
inch = in.nextInt();
```

接下来，我们将英寸数乘以 2.54（因为 1 英寸相当于 2.54 厘米）并显示结果：

```
cm = inch * 2.54;
System.out.print(inch + " in = ");
System.out.println(cm + " cm");
```

这些代码可以正确运行，但存在一个小问题：其他程序员看到这些代码时，可能不明白 2.54 是怎么来的。为方便其他程序员（还有未来的你），更好的做法是将这个值赋给一个变量，并给这个变量指定一个有意义的名称。我们将在 3.5 节中对此进行演示。

3.5 字面量和常量

在程序中，2.54（或 `" in ="`）这样的值被称为*字面量*（literal）。一般而言，使用字面量没什么错，但如果在表达式中使用 2.54 这样的数字，却不作任何解释的话，代码将难以理解。另外，如果同样的值出现多次，且以后可能需要修改，那么代码将难以维护。

这样的值有时被称为*魔幻数字*（magic number，这里的“魔幻”可不是褒义的），一种很好的做法是像以下这样将魔幻数字赋给变量，并给变量指定有意义的名称：

```
double cmPerInch = 2.54;
cm = inch * cmPerInch;
```

这个版本更容易理解，而且不那么容易出错，但还是存在一个问题，那就是变量是可变的，而 1 英寸对应的厘米数是不变的。一旦给 `cmPerInch` 赋值，就再也不应该修改。Java 提供了实施这种规则的语言特性——关键词 `final`。

```
final double CM_PER_INCH = 2.54;
```

将变量声明为 `final` 意味着对其进行初始化后，就不能重新赋值了。如果你试图这样做，编译器就会报错。声明为 `final` 的变量被称为*常量*（constant）；根据约定，常量名全部大写，且单词间用下划线（_）连接。

3.6 设置输出的格式

使用 `print` 或 `println` 输出 `double` 值时，最多显示 16 位小数：

```
System.out.print(4.0 / 3.0);
```

结果如下：

```
1.3333333333333333
```

这可能比你想要的要多。`System.out` 提供了另一个方法 `printf`，让你对输出格式有更大的控制权，`printf` 中的 f 指的是“格式化”。以下是一个示例：

```
System.out.printf("Four thirds = %.3f", 4.0 / 3.0);
```

括号中的第一个值为*格式字符串*（format string），指定了输出的显示方式。上述的格式字符串中包含普通文本，普通文本的后面是*格式说明符*（format specifier）——以百分号打头的特殊序列。格式说明符 `%.3f` 指定接下来的值应显示为浮点数，并舍入到三位小数。上述代码的结果如下：

```
Four thirds = 1.333
```

格式字符串可包含任意数目的格式说明符，下面的格式字符串就包含了两个格式说明符：

```
int inch = 100;
double cm = inch * CM_PER_INCH;
System.out.printf("%d in = %f cm\n", inch, cm);
```

结果如下：

```
100 in = 254.000000 cm
```

与 `print` 一样，`printf` 也不在末尾换行；所以格式字符串通常以换行符结尾。

格式说明符 `%d` 用于显示整数值，其中的 d 表示 decimal（十进制）。值依次与格式说明符配对，因此，用于 `inch` 的格式说明符为 `%d`，用于 `cm` 的格式说明符为 `%f`。

学习格式字符串相当于学习 Java 中的一种子语言；涉及的选项很多，细节可能让人不可重负。表 3-1 列出了一些常用的格式说明符，旨在让你大致地了解其中的工作原理；更多细节请参阅 `java.util.Formatter` 的相关文档。要想找到有关 Java 类的文档，最简单的方法是在网上搜索 Java 和类名。

表3-1：格式说明符示例

`%d`	十进制整数	12345
`%08d`	添加前导零，确保显示的值至少包含 8 位	00012345
`%f`	浮点数	6.789000
`%.2f`	舍入到两位小数	6.79

3.7 厘米到英寸的转换

现在假设有一个以厘米为单位的值，我们想将其转换为与之最接近的英寸数。你可能很想这样编写代码：

```
inch = cm / CM_PER_INCH;  // 语法错误
```

但这将导致编译错误，会出现类似于“Bad types in assignment: from double to int”这样的错误消息。这是因为右边是浮点数，而左边是整数变量。

要将浮点值转换为整数，最简单的方式是使用*类型转换*（type cast），类型转换因将值从一种类型塑造或铸造成另一种类型而得名。类型转换的语法是将类型名放在括号内，并将其用作运算符。

```
double pi = 3.14159;
int x = (int) pi;
```

运算符 (int) 会将它后面的值转换为整数。在这个示例中，x 将被设置为 3。与整数除法一样，转换为整数时总是向下圆整，即便小数部分为 0.999999（或 -0.999999）也是如此。换言之，将直接丢弃小数部分。

类型转换的优先级高于算术运算。在下面的示例中，先将 pi 的值转换为整数，然后再执行乘法运算，因此结果为 60.0，而不是 62.0。

```
double pi = 3.14159;
double x = (int) pi * 20.0;
```

请务必牢记这一点。下面的代码演示了如何将厘米数转换为英寸数：

```
inch = (int) (cm / CM_PER_INCH);
System.out.printf("%f cm = %d in\n", cent, inch);
```

转换运算符后面的括号使得除法运算先执行，然后再进行类型转换。因此除法运算的结果将向下圆整；我们将在下一章介绍如何将浮点数圆整为与之最接近的整数。

3.8　求模运算符

现在我们再进一步：假设你有一个以英寸为单位的值，并且想将其转换为英尺数和英寸数。为此，需要除以 12（1 英尺对应 12 英寸），并将余数记录下来。

本书前面已经介绍过除法运算符（/），用于计算两个数的商。如果两个操作数都为整数，那么它将执行整数除法。Java 还提供了*求模*（modulus）运算符（%），用于计算两个数相除的余数。

可像以下这样用除法和求模运算来将英寸数转换为英尺数和英寸数：

```
quotient = 76 / 12;   // 除法
remainder = 76 % 12;  // 求模
```

第 1 行的结果为 6，第 2 行读作“76 与 12 的模”，结果为 4。因此，76 英寸相当于 6 英尺 4 英寸。

求模运算符看起来像百分号，但其实可以将其视为除号（÷）向左旋转 90 度的结果。

求模运算符很有用，例如，可以检查一个数能否被另一个数整除：如果 x % y 的结果

为零，就说明 x 能够被 y 整除。可用求模运算来提取数字中的某些位：x % 10 的结果为 x 的个位，而 x % 100 的结果为最后两位。另外，很多加密算法都大量地使用了求模运算符。

3.9 整合

至此，你应该对 Java 有足够的了解，能够编写解决日常问题的程序了。你知道如何导入 Java 库中的类、如何创建 Scanner 对象、如何从键盘获取输入、如何用 printf 设置输出的格式以及如何执行整数除法和求模运算。现在让我们结合这些知识，编写一个完整的程序：

```
import java.util.Scanner;

/**
 * 将厘米数转换为英尺数和英寸数
 */
public class Convert {

    public static void main(String[] args) {
        double cm;
        int feet, inches, remainder;
        final double CM_PER_INCH = 2.54;
        final int IN_PER_FOOT = 12;
        Scanner in = new Scanner(System.in);

        // 提示用户输入值并读取这个值
        System.out.print("Exactly how many cm? ");
        cm = in.nextDouble();

        // 转换并输出结果
        inches = (int) (cm / CM_PER_INCH);
        feet = inches / IN_PER_FOOT;
        remainder = inches % IN_PER_FOOT;
        System.out.printf("%.2f cm = %d ft, %d in\n",
                          cm, feet, remainder);
    }
}
```

虽然没有要求，但所有的变量和常量都是在 main 方法的开头声明的。这种做法让人更容易获悉这些变量和常量的类型以及算法涉及了哪些数据。

为提高可读性，可用一个空行将算法的主要步骤分隔开，且每个主要步骤都以注释打头。这个程序还包含文档注释（/**），我们将在下一章中更详细地介绍。

很多算法（包括程序 Convert）都结合使用了除法和求模运算。在程序 Convert 中，这两种运算使用的除数相同，都是 IN_PER_FOOT。

常见的编程风格是将过长（超过 80 个字符）的语句分成多行，以便阅读时不用水平滚动。

3.10　Scanner类的bug

在有了一些使用 Scanner 的经验后，有必要提醒你一下，它存在一个出人意料的行为。下面的代码片段让用户输入姓名和年龄：

```
System.out.print("What is your name? ");
name = in.nextLine();
System.out.print("What is your age? ");
age = in.nextInt();
System.out.printf("Hello %s, age %d\n", name, age);
```

其输出可能类似于以下这样：

```
Hello Grace Hopper, age 45
```

先读取 String 再读取 int 时，一切都正常，但如果先读取 int 再读取 String，将发生怪异的事情。

```
System.out.print("What is your age? ");
age = in.nextInt();
System.out.print("What is your name? ");
name = in.nextLine();
System.out.printf("Hello %s, age %d\n", name, age);
```

如果尝试运行上述示例代码，你将发现它根本不允许输入姓名，而在你输入年龄后会直接显示输出：

```
What is your name? Hello , age 45
```

要想明白其中的原因，你必须知道的是，Scanner 并不像我们那样将输入视为多行；相反，它获得的是一个如图 3-3 所示的“字符流”。

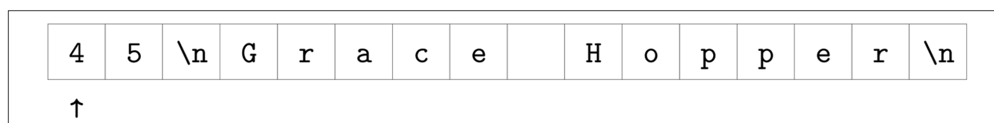


图 3-3：Scanner 眼中的字符流

其中的箭头指向 Scanner 将读取的下一个字符。调用 nextInt 时，它会不断读取字符，直到遇到非数字字符。图 3-4 显示了调用 nextInt 后流的状态。



图 3-4：调用 nextInt 后的字符流

此时，nextInt 返回 45。接下来，程序显示提示 What is your name?，并调用 nextLine；这个方法会不断读取字符，直到遇到换行符。然而，由于下一个字符就是换行符，因此 nextLine 返回一个空字符串（""）。

为解决这个问题，需要在 nextInt 后面多调用一次 nextLine。

```
System.out.print("What is your age? ");
age = in.nextInt();
in.nextLine();  // 读取换行符
System.out.print("What is your name? ");
name = in.nextLine();
System.out.printf("Hello %s, age %d\n", name, age);
```

读取独占一行的 int 或 double 值时，经常需要使用技巧：先读取数字，再读取当前行余下的全部内容（仅仅是一个换行符）。

3.11 术语表

- 包
  一组彼此相关的类。

- 地址
  值在计算机内存中的位置，通常用十六进制整数表示。

- 库
  可在其他程序中使用的一系列包和类。

- 导入语句
  让程序能够使用其他包中定义的类的语句。

- 标记
  程序的基本元素，如单词、空格、符号或数字。

- 字面量
  在源代码中出现的值，例如，"Hello" 是一个字符串字面量，而 74 是一个整数字面量。

- 魔幻数字
  表达式中没有任何解释的数字，通常应将其替换为常量。

- 常量
  声明为 `final` 的变量，其值不可修改。

- 格式字符串
  传递给 `printf` 以便指定输出格式的字符串。

- 格式说明符
  以百分号开头的特殊编码，指定了相应值的数据类型和格式。

- 类型转换
  从一种类型明确地转换为另一种类型的操作。在 Java 中表现为用括号括起的类型名，如 `(int)`。

- 求模
  一种运算符，用于计算两个整数相除的余数。在 Java 中用百分号表示。例如，`5 \% 2` 的结果为 `1`。

3.12 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch03 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

如果你还没有阅读 A.3 节，那么现在正是阅读的好时机。该节介绍了命令行界面，这是与计算机交互的强大而高效的一种方式。

练习3-1

使用 `printf` 时，Java 编译器不会检查其中的格式字符串。请尝试用 `%f` 显示一个类型为 `int` 的值，并看看结果将会如何。用 `%d` 显示 `double` 值呢？指定两个格式说明符，却只提供一个值，又会发生什么呢？

练习3-2

编写一个程序，将摄氏温度转换为华氏温度。这个程序应：(1) 提示用户输入摄氏温度值；(2) 从键盘读取一个 `double` 值；(3) 计算结果；(4) 将输出设置为包含一位小数。例如，它应显示 `"24.0 C=75.2 F"` 这样的输出。

下面是转换公式，请注意，千万不要用整数除法！

$$F = C \times \frac{9}{5} + 32$$

练习3-3

编写一个程序，将秒数转换为小时数、分钟数和秒数。这个程序应：(1) 提示用户输入秒数；(2) 从键盘读取一个整数；(3) 计算结果；(4) 用 `printf` 显示输出。例如，它应显示 `"5000 seconds = 1 hours, 23 minutes, and 20 seconds"` 这样的输出。

提示：可使用求模运算符。

练习3-4

这个练习的目标是编写一个“猜数”游戏，其输出应类似于以下这样：

```
I'm thinking of a number between 1 and 100
(including both). Can you guess what it is?
Type a number: 45
Your guess is: 45
The number I was thinking of is: 14
You were off by: 31
```

要想生成随机数，可使用 `java.util` 中的 `Random` 类，其工作原理如下：

```
import java.util.Random;

public class GuessStarter {

    public static void main(String[] args) {
        // 生成一个随机数
        Random random = new Random();
        int number = random.nextInt(100) + 1;
        System.out.println(number);
    }
}
```

与本章介绍的 `Scanner` 类一样，要想使用 `Random` 类，必须先导入。另外，与创建 `Scanner` 对象一样，必须用 `new` 运算符创建一个 `Random` 对象（随机数生成器）。

然后就可以用方法 `nextInt` 来生成随机数了。在上面的示例中，`nextInt(100)` 的结果是一个 0~99（闭区间）的数字。通过将这个结果加 1，将得到一个 1~100（闭区间）的数字。

(1) `GuessStarter` 的定义位于文件 GuessStarter.java 中，而这个文件位于本书代码仓库的目录 ch03 中。
(2) 编译并运行这个程序。
(3) 修改这个程序，提示用户输入一个数字，再用 `Scanner` 读取一行用户输入。编译并测试这个程序。
(4) 将用户输入作为整数读取，并显示结果。再次编译并测试这个程序。
(5) 计算并显示用户猜测的数字和生成的随机数之间的差。

第 4 章

void方法

到目前为止，我们编写的程序都很短，只包含一个类和一个方法（`main`）。我们将在本章演示如何将较长的程序组织成多个方法和类，还将介绍 `Math` 类，它提供了执行常见数学运算的方法。

4.1　Math类的方法

学习数学时，你可能见过 sin 和 log 这样的函数，还学习过如何计算像 sin(π/2) 和 log(1/*x*) 这样的表达式的值：先计算括号内的表达式，它们被称为函数的实参（argument），然后计算函数本身，可能还要用到计算器。

计算 log(1/sin(π/2)) 这样更复杂的表达式时，可重复执行这个过程：先计算最里面的函数的实参，再计算函数本身，依此类推。

Java 库包含一个 `Math` 类，它提供了执行常见数学运算的方法。这个类位于 `java.lang` 包中，因此无需导入。可像下面这样使用或调用（invoke）`Math` 类的方法：

```
double root = Math.sqrt(17.0);
double angle = 1.5;
double height = Math.sin(angle);
```

第 1 行将 `root` 设置为 17 的平方根；第 3 行计算 1.5（变量 `angle` 的值）的正弦。

三角函数 sin、cos 和 tan 的实参应以弧度为单位。要想将度数转换为弧度数，可将其除以 180 再乘以 π。好在 `Math` 类提供了一个名为 `PI` 的 `double` 常量，它包含 π 的近似值：

```
double degrees = 90;
double angle = degrees / 180.0 * Math.PI;
```

请注意，常量名 PI 的字母全都是大写；Java 根本不知道 Pi、pi 和 pie 为何物。另外，PI 是一个变量而不是方法的名称，因此它后面不带括号。常量 Math.E 亦是如此，它包含欧拉数的近似值。

在度数和弧度数之间进行转换是一种常见的运算，因此 Math 类提供了执行这种运算的方法。

```
double radians = Math.toRadians(180.0);
double degrees = Math.toDegrees(Math.PI);
```

另一个很有用的方法是 round，它将浮点数圆整为最接近的整数，并将其作为 long 值返回。long 类似于 int，但可表示的值更大。更具体地说，int 长 32 位，可存储的最大值为 $2^{31} - 1$——约为 20 亿；long 长 64 位，可存储的最大值为 $2^{63} - 1$——约为 9×10^{18}。

```
long x = Math.round(Math.PI * 20.0);
```

结果为 63（这是将 62.8319 向上圆整得到的）。

请花点时间阅读 Math 类的这些方法和其他方法的文档。要想查找有关 Java 类的文档，最简单的方法是在网上搜索 Java 和类名。

4.2　再谈组合

与数学函数一样，Java 方法也是可以组合的，这意味着可在一个表达式中包含另一个表达式。例如，可将任何表达式用作方法的实参：

```
double x = Math.cos(angle + Math.PI / 2.0);
```

这条语句将 Math.PI 除以 2，再将结果与 angle 相加，然后计算得到的和的余弦。还可将一个方法的结果用作另一个方法的实参：

```
double x = Math.exp(Math.log(10.0));
```

在 Java 中，方法 log 总是以 e 为底，因此这条语句计算以 e 为底的 10 的对数，再将结果作为指数计算 e 的相应次幂，然后将得到的结果赋给变量 x。

Math 类的有些方法接受多个实参，例如，Math.pow 接受两个实参，并计算第一个实参的第二个实参次幂。下面的这行代码将值 1024.0 赋给变量 x：

```
double x = Math.pow(2.0, 10.0);
```

在用 Math 类的方法时，遗漏 Math 是一种常见的错误。例如，如果试图调用 pow(2.0,

10.0)，将出现类似于以下的错误消息：

```
File: Test.java [line: 5]
Error: cannot find symbol
  symbol: method pow(double,double)
  location: class Test
```

消息 cannot find symbol 令人迷惑，但最后一行提供了有用的线索：编译器试图在调用方法 pow 的类（Test）中查找它。如果没有指定类名，编译器将在当前类中查找。

4.3　添加方法

此时你应该已经猜到了，在一个类中可定义多个方法，如下例所示：

```
public class NewLine {

    public static void newLine() {
        System.out.println();
    }

    public static void main(String[] args) {
        System.out.println("First line.");
        newLine();
        System.out.println("Second line.");
    }
}
```

类名为 NewLine。根据约定，类名的首字母要大写。NewLine 包含两个方法：newLine 和 main。别忘了，Java 区分大小写，因此 NewLine 和 newLine 不是一回事。

方法名的首字母应小写，并采用“骆驼拼写法”，即类似于 jammingWordsTogetherLikeThis 这样的格式。可给方法指定任何名称，但 main 和其他 Java 关键词除外。

newLine 和 main 是公有的，这意味着可在其他类中调用。它们都是静态的，那么静态是什么意思呢？现在暂时无法解释。另外，它们的返回类型都是 void，这意味着它们不会像 Math 类的方法那样返回结果。

方法名后面的括号包含了一个变量列表，这些变量被称为*形参*（parameter），用于存储方法的实参。main 只有一个名为 args 的形参，类型为 String[]，这意味着调用这个 main 方法时，必须提供一个字符串数组（将在本书的后面介绍）。

由于 newLine 没有形参，所以调用时不需要提供实参，如在 main 方法中的调用所示。另外，由于 newLine 和 main 位于同一个类中，因此在 main 中调用 newLine 时无需指定类名。

这个程序的输出如下：

```
First line.

Second line.
```

注意，输出行之间有空行。如果要想让输出行相距得更远，可多次调用方法 `newLine`：

```
public static void main(String[] args) {
    System.out.println("First line.");
    newLine();
    newLine();
    newLine();
    System.out.println("Second line.");
}
```

我们也可以再编写一个显示三个空行的方法：

```
public static void threeLine() {
    newLine();
    newLine();
    newLine();
}

public static void main(String[] args) {
    System.out.println("First line.");
    threeLine();
    System.out.println("Second line.");
}
```

可以多次调用同一个方法，还可以在一个方法中调用另一个方法。在这个示例中，方法 `main` 调用了方法 `threeLine`，而方法 `threeLine` 调用了 `newLine`。

初学者常常对不厌其烦地创建新方法心存疑惑。这样做的原因很多，这个示例说明了其中的几个原因。

- 通过创建新方法，你可以给一组语句指定名称，从而让代码更容易阅读和理解。
- 引入新方法可消除重复的代码，从而缩小程序的规模。例如，要显示 9 个空行，可调用 `threeLine` 三次。
- 一种常见的问题解决技巧是将任务划分成子问题。方法让你能够只专注于子问题，然后再将方法组合成完整的解决方案。

4.4 执行流程

将 4.3 节中的代码组合起来可以得到类似于下面这样的完整程序：

```
public class NewLine {

    public static void newLine() {
        System.out.println();
```

```
    }

    public static void threeLine() {
        newLine();
        newLine();
        newLine();
    }

    public static void main(String[] args) {
        System.out.println("First line.");
        threeLine();
        System.out.println("Second line.");
    }
}
```

阅读包含多个方法的类定义时，你可能很想按照从头到尾的顺序阅读。但这样做很可能让你感到迷惑，因为程序的*执行流程*（flow of excution）并不是这样的。

不管 `main` 方法位于源代码文件的什么地方，程序总是从这个方法的第一条语句开始执行。语句按顺序以每次一条的方式执行，直到遇到方法调用。我们可将方法调用视为改道：不是直接执行下一条语句，而是跳转到被调用的方法的第一行，执行完这个方法的全部语句后，再回到离开的地方继续执行。

这好像很简单，但别忘了，一个方法可以调用另一个方法。执行 `main` 方法时，中途离开去执行 `threeLine` 的语句；执行 `threeLine` 时，又中途离开去执行 `newLine`；而 `newLine` 调用 `println`，导致再次改道。

好在 Java 擅于跟踪当前的运行方法，因此，`println` 执行完毕后，它会回到离开 `newLine` 的地方；`newLine` 执行完毕后，回到离开 `threeLine` 的地方；而 `threeLine` 执行完毕后，又回到离开 `main` 的地方。

总之，阅读程序时，不要按照从头到尾的顺序阅读，而应按执行流程阅读。

4.5　形参和实参

前面使用的有些方法需要实参，实参是调用方法时提供的值。例如，要计算一个数字的正弦就必须提供这个数字，因此，`sin` 接受一个 `double` 实参。要显示一条消息就必须提供这条消息，因此，`println` 接受一个 `String` 实参。

在使用方法时提供的是实参，在编写方法时指定的是形参。形参列表指定了必须提供哪些实参，以下是一个示例：

```
public class PrintTwice {

    public static void printTwice(String s) {
        System.out.println(s);
```

```
            System.out.println(s);
        }

        public static void main(String[] args) {
            printTwice("Don't make me say this twice!");
        }
    }
```

printTwice 包含了一个名为 s，类型为 String 的形参。调用 printTwice 时，必须提供一个类型为 String 的实参。

方法执行前，实参会赋给形参。在这个示例中，实参 "Don't make me say this twice!" 被赋给形参 s。

这个过程被称为*参数传递*（parameter passing），因为值从方法外传到了方法内。实参可以是任何表达式，因此，如果声明了一个 String 变量，就可将其用作实参：

```
String argument = "Never say never.";
printTwice(argument);
```

用作实参的值必须与形参的类型相同，例如，如果你试图像下面这样做：

```
printTwice(17);  // 语法错误
```

将出现类似于下面这样的错误消息：

```
File: Test.java [line: 10]
Error: method printTwice in class Test cannot be applied
       to given types;
  required: java.lang.String
  found: int
  reason: actual argument int cannot be converted to
          java.lang.String by method invocation conversion
```

在有些情况下，Java 能够自动将实参从一种类型转换为另一种类型。例如，Math.sqrt 需要接受一个 double 实参，但如果你调用了 Math.sqrt(25)，整数值 25 将自动转换为浮点值 25.0。但就 printTwice 而言，Java 并不能（或不会）将提供给它的整数 17 转换为 String。

形参和其他变量只存在于当前的方法中；在 main 中，没有 s 这样的变量，如果试图在这个方法中使用它，将导致编译错误。同样，printTwice 中没有 args 变量，这个变量位于 main 中。

鉴于变量只在定义它的方法中存在，因此它们常被称为*局部变量*（local varible）。

4.6 多个形参

下面是一个接受两个形参的方法：

```
public static void printTime(int hour, int minute) {
    System.out.print(hour);
    System.out.print(":");
    System.out.println(minute);
}
```

你可能很想将上述的形参列表写成下面这样：

```
public static void printTime(int hour, minute) {
    ...
```

但这种格式（省略第二个 int）只适用于变量声明；在形参列表中，必须分别指定每个变量的类型。

要调用这个方法，必须提供两个整数实参：

```
int hour = 11;
int minute = 59;
printTime(hour, minute);
```

常见的错误是像下面这样声明实参的类型：

```
int hour = 11;
int minute = 59;
printTime(int hour, int minute);  // 语法错误
```

这是一种语法错误：在编译器看来，int hour 和 int minute 是变量声明，而不是表达式。如果将整数字面量用作实参，则不能声明它们的类型：

```
printTime(int 11, int 59); // 语法错误
```

4.7 栈图

将 4.6 节的代码片段组合起来可以得到完整的类定义，如下：

```
public class PrintTime {

    public static void printTime(int hour, int minute) {
        System.out.print(hour);
        System.out.print(":");
        System.out.println(minute);
    }

    public static void main(String[] args) {
        int hour = 11;
        int minute = 59;
        printTime(hour, minute);
    }
}
```

printTime 有两个形参——hour 和 minute。main 有两个变量，也分别名为 hour 和 minute。虽然名称相同，但这些变量是不同的变量：printTime 中的 hour 和 main 中的 hour 指向不同的存储位置，可以有不同的值。

例如，你可以像下面这样调用 printTime：

```
int hour = 11;
int minute = 59;
printTime(hour + 1, 0);
```

调用这个方法前，Java 会先计算实参的值，这里分别为 12 和 0，然后将这些值赋给形参。在 printTime 中，hour 的值为 12，而不是 11，而 minute 的值为 0，而不是 59。另外，即便 printTime 修改了其形参的值，但不会影响 main 中的变量。

要想对程序的方方面面进行跟踪，一种方法是绘制栈图（stack diagram）。栈图是一种显示方法调用的状态图（参见 2.3 节）。对于每一个方法，栈图中都有一个名为栈帧（frame）的方框，其中包含了这个方法的形参和变量。方法名位于栈帧的外面，而变量和形参则位于栈帧内。

与状态图一样，栈图也显示了变量和方法在特定时点的状态。图 4-1 是方法 printTime 刚执行时的栈图。

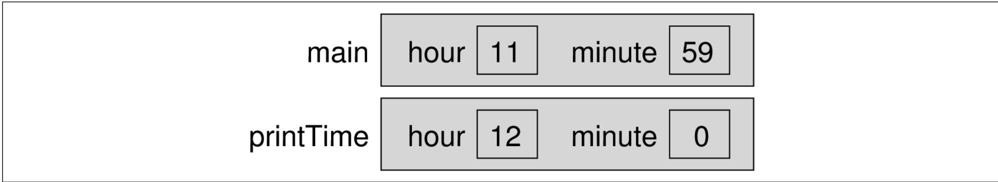


图 4-1：PrintTime 的栈图

4.8 阅读文档

Java 的优点之一是自带了庞大的类库和方法。但在使用之前，可能还必须阅读文档，而这并非总是那么容易。

例如，咱们来看看 3.2 节中使用的 Scanner 类文档。要找到这个文档，可在网上搜索 Java Scanner，图 4-2 是该文档页面的屏幕截图。

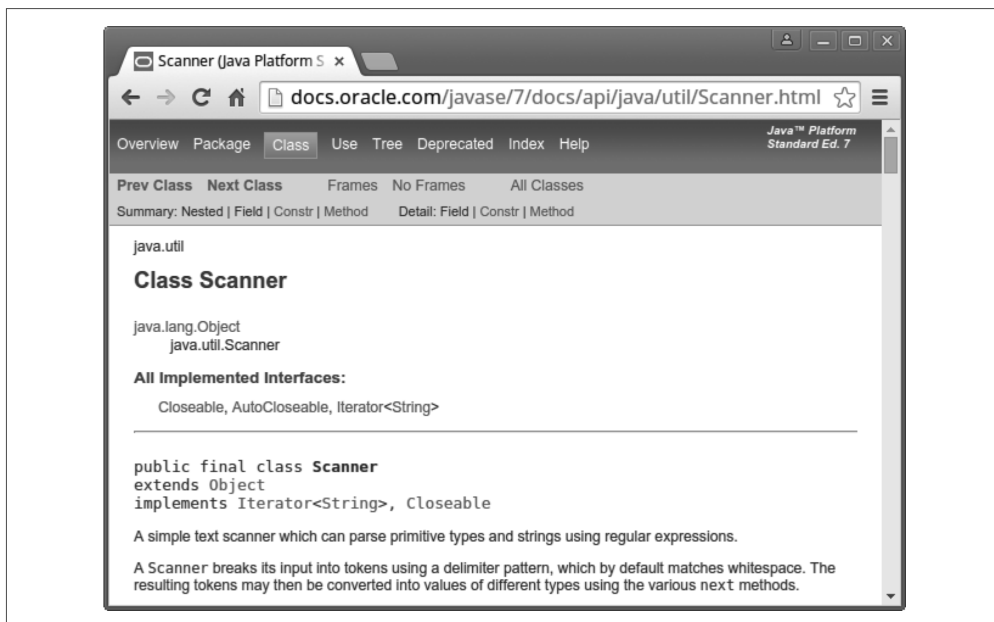


图 4-2：Scanner 文档的屏幕截图

其他类的文档格式与此类似。第 1 行是类所属的包，如 java.util；第 2 行是类名。截屏中的 All Implemented Interfaces 列举了部分 Scanner 能够做的事情，这里不打算做更深入的介绍。

文档的下一部分是描述，阐述了当前类的用途，还包含了使用示例。这些内容可能难以理解，因为其中使用了你还没有学过的术语，但其中的示例经常很有用。要开始使用新类，一种不错的方式是将文档中的示例粘贴到测试文件中，看看它们能否编译并运行。

有个示例演示了如何用 Scanner 从 String 而不是 System.in 读取输入：

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
```

描述、代码示例和其他细节的后面是以下几个表格：

- Constructor summary

  创建或构造 Scanner 对象的方式

- Method summary

  Scanner 提供的方法列表

- Constructor detail

  更多与 Scanner 对象创建方式有关的信息

- Method detail
  有关各个方法的详细信息

例如，下面是 `nextInt` 的摘要信息：

```
public int nextInt()
Scans the next token of the input as an int.
```

第 1 行是方法的特征标（signature），它指定了方法的名称、形参（无）和返回类型（`int`）；第 2 行简单地描述了这个方法的功能。

表格 Method detail 更详细地阐述了这个方法：

```
public int nextInt()
Scans the next token of the input as an int.

An invocation of this method of the form nextInt() behaves in
exactly the same way as the invocation nextInt(radix), where
radix is the default radix of this scanner.

Returns:
the int scanned from the input
Throws:
InputMismatchException - if the next token does not match
    the Integer regular expression, or is out of range
NoSuchElementException - if input is exhausted
IllegalStateException - if this scanner is closed
```

其中的 Returns 部分描述了这个方法成功时返回的结果，而 Throws 部分描述了可能发生的错误及其引发的异常。

要想熟练地阅读文档并就哪些部分可以忽略作出准确的判断，可能需要一段时间的学习，但这样的付出是值得的。知道 Java 库有哪些类可避免重复劳动，只需阅读少量的文档就可避免繁重的调试工作。

4.9 编写文档

受益于优秀文档的同时，应该编写优秀的文档来作为回报。Java 语言提供了一项很好的功能，即可以在源代码中嵌入文档。这让你能够在编写代码的同时编写文档，同时，更容易在修改代码时确保文档与代码一致。

可用工具 Javadoc 自动提取包含在源代码中的文档，并生成格式良好的 HTML。这个工具包含在标准 Java 开发环境中，被大家广泛使用。事实上，Java 库的在线文档就是用 Javadoc 生成的。

Javadoc 扫描源代码文件以查找格式特殊的*文档注释*——也称为 Javadoc 注释。这种注释

以 /**（两个星号）打头，并以 */（一个星号）结尾，位于它们之间的所有内容都被视为文档。

下面的类定义包含两条 Javadoc 注释，其中一条是针对类的，另一条是针对方法 main 的：

```
/**
 * 演示print和println的示例程序
 */
public class Goodbye {

    /**
     * 打印问候语
     */
    public static void main(String[] args) {
        System.out.print("Goodbye, "); // 请注意其中的空格
        System.out.println("cruel world");
    }
}
```

类注释阐述了类的目的，而方法注释阐述了方法的作用。

注意，这个示例还包含了一条以 // 打头的内嵌注释。一般而言，内嵌注释是对程序复杂部分进行诠释的短语，旨在帮助其他程序员理解和维护源代码。

相反，Javadoc 注释更长，通常是完整的句子。它们阐述每个方法的功能，但不涉及其工作原理的细节，旨在让人无需查看源代码就能使用这些方法。

要想让源代码易于理解，合适的注释和文档是必不可少的。另外别忘了，对于你编写的代码，未来阅读得最多的人就是你自己，届时你定将庆幸自己编写了优秀的文档。

4.10 术语表

- 实参
  调用方法时提供的值，其类型必须与相应形参的类型相同。

- 调用
  执行方法。

- 形参
  运行方法所需要的信息。形参也是变量，包含值和类型。

- 执行流程
  Java 执行方法和语句的顺序，这种顺序并不一定是从上到下、从左到右的。

- 参数传递
  将实参的值赋给形参变量的过程。

- 局部变量

  在方法内声明的变量；在所属方法外不能访问。

- 栈图

  各个方法中的变量的图形化表示。执行流程中的方法调用按从上到下的顺序堆叠。

- 栈帧

  在栈图中表示特定方法中的变量和形参及其当前值的部分。

- 特征标

  方法的第一行，定义了方法的名称、返回类型和形参。

- Javadoc

  读取 Java 源代码并生成 HTML 格式文档的工具。

- 文档

  描述类或方法用法的注释。

4.11 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch04 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

如果你还没有阅读 A.4 节，那么现在正是阅读的好时机。该节介绍了对接受用户输入并显示输出的程序进行测试的一种高效方式。

练习4-1

这个练习的意义在于阅读包含多个方法的程序的代码，并确保你明白其执行流程。

(1) 下面程序的输出是什么？务必准确地指出哪些地方有空格以及在哪些地方换行了。

  提示：先口头描述 `ping` 和 `baffle` 被调用时会做什么。

(2) 绘制一个状态图，显示 `ping` 首次被调用时程序的状态。

(3) 如果在方法 `ping` 的最后调用 `baffle()`，结果将会如何？（我们将在下一章介绍其中的原因。）

```
public static void zoop() {
    baffle();
    System.out.print("You wugga ");
    baffle();
}

public static void main(String[] args) {
    System.out.print("No, I ");
```

```
        zoop();
        System.out.print("I ");
        baffle();
    }

    public static void baffle() {
        System.out.print("wug");
        ping();
    }

    public static void ping() {
        System.out.println(".");
    }
```

练习4-2

这个练习旨在确保你明白如何编写和调用接受参数的方法。

(1) 编写方法 `zool` 的第一行，这个方法包含三个形参：一个 `int` 形参和两个 `String` 形参。

(2) 编写调用 `zool` 的代码，它传递的实参为值 `11`、你养的第一个宠物的名字以及你小时候居住的街道。

练习4-3

这个练习的目的在于将之前编写的代码封装到一个接受参数的方法中。做这个练习前，必须先完成练习 2-2。

(1) 编写一个名为 `printAmerican` 的方法，让其接受参数 day、date、month 和 year，并以美国格式显示。

(2) 对这个方法进行测试：在 `main` 中调用该方法并传递合适的实参。输出应类似于以下这样（只是日期可能不同）：

```
Saturday, July 22, 2015
```

(3) 确定方法 `printAmerican` 正确无误后，再编写一个以欧洲格式显示日期的方法，并将其命名为 `printEuropean`。

第5章 条件和逻辑

不管输入如何，前几章中的程序每次运行时做的事情几乎相同。进行更复杂的计算时，通常需要程序根据输入作出反应，即检查特定的条件并生成相应的结果。本章将介绍能让程序作出决策的功能：一种名为 `boolean` 的数据类型、表示逻辑的运算符以及 `if` 语句。

5.1 关系运算符

关系运算符（relational operator）用于检查条件，如两个值是否相等或一个值是否大于另一个值。以下的表达式演示了关系运算符的用法：

```
x == y          // x等于y
x != y          // x与y不相等
x > y           // x大于y
x < y           // x小于y
x >= y          // x大于或等于y
x <= y          // x小于或等于y
```

关系运算符的结果为 `true` 或 `false` 这两个特殊值中的一个。这些值属于 `boolean` 数据类型；事实上，它们是仅有的两个 `boolean` 值。

你可能熟悉这些运算，但注意，表示这些运算时，Java 使用的运算符不同于数学中使用的符号（如 =、≠和≤）。一种常见的错误是使用单个等号（`=`）而不是两个（`==`）。别忘了，`=` 是赋值运算符，而 `==` 是一个比较运算符。另外，没有诸如 `=<` 和 `=>` 这样的 Java 运算符。

关系运算符的两边必须兼容，例如，表达式 `5 < "6"` 是非法的，因为 `5` 是一个 `int`，而 `"6"` 是一个 `String`。比较不同类型的数值时，Java 应用前面介绍过的赋值运算符的转换规则。

例如，计算表达式 `5 < 6.0` 时，Java 自动将 5 转换为 `5.0`。

大多数的关系运算符都不可用于字符串，但令人迷惑的是，`==` 和 `!=` 可以，只是它们的行为并非你预期的那样，我们将在后面有所介绍。在此之前，不要将它们用于字符串，而应使用方法 `equals`：

```
String fruit1 = "Apple";
String fruit2 = "Orange";
System.out.println(fruit1.equals(fruit2));
```

`fruit1.equals(fruit2)` 的结果为 `boolean` 值 `false`。

5.2 逻辑运算符

Java 提供了三个*逻辑运算符*（logical operator）：`&&`、`||` 和 `!`，分别表示与、或、非。这些运算符的结果与其在英语中的含义类似。

例如，如果 `x` 大于 0 且小于 10，那么 `x > 0 && x < 10` 的结果为 `true`；对于表达式 `evenFlag || n \% 3 == 0`，只要其中一个条件为 true，即如果 `evenFlag` 为 true 或数字 `n` 能被 3 整除，那么这个表达式的结果就为 `true`。最后，运算符 `!` 对 `boolean` 表达式求反，因此，如果 `evenFlag` 不为 `true`，则 `!evenFlag` 为 `true`。

逻辑运算符仅在必要时才计算第二个表达式的值。例如，`true || anything` 的结果在任何情况下都为 `true`，因此 Java 无需计算表达式 `anything` 的值。同样，`false && anything` 的结果在任何情况下都为 `false`。在可能的情况下忽略第二个操作数被称为*短路*（short circuit）求值，可与电路类比。短路求值可节省时间，在 `anything` 的值需要很长时间才能计算出时尤其如此。如果 `anything` 可能出现问题的话，这还可以避免不必要的错误。

如果你曾必须对包含逻辑运算符的表达式求反，以后也很可能遇到这样的情况。在这种情况下，德·摩根定律（De Morgan's laws）可以提供帮助：

- `!(A && B)` 与 `!A || !B` 等价
- `!(A || B)` 与 `!A && !B` 等价

上述列表表明，要对逻辑表达式求反，可分别对每一项求反，并使用相反的运算符。运算符 `!` 的优先级比 `&&` 和 `||` 高，因此不需要将 `!A` 和 `!B` 分别放在括号中。

德·摩根定律也适用于关系运算符。在这种情况下，对每一项求反意味着使用“相反”的运算符：

- `!(x < 5 && y == 3 )` 与 `x > =5 || y! = 3` 等价
- `!(x >= 1 || y! = 7 )` 与 `x < 1 && y == 7` 等价

将这些示例大声地朗读出来可能会有所帮助。例如，“如果不希望 x 小于 5，且不希望 y 为 3，就意味着 x 必须大于或等于 5，且 y 不能为 3。”

5.3 条件语句

为了编写有用的程序，几乎都需要检查添加并采取相应的措施。条件语句（conditional statement）提供了这样的功能。if 语句是 Java 中最简单的条件语句：

```
if (x > 0) {
    System.out.println("x is positive");
}
```

括号内的表达式被称为条件，如果它为 true，那么将执行大括号内的语句，否则将跳过这个代码块。括号内的条件可以是任何 boolean 表达式。

还有一种包含两种可能性（分别由 if 和 else 标识）的条件语句。这些可能性称为分支（branch），由条件决定将执行哪个分支：

```
if (x % 2 == 0) {
    System.out.println("x is even");
} else {
    System.out.println("x is odd");
}
```

如果 x 除以 2 的余数为 0，则 x 为偶数，因此上述代码片段显示相应的消息。如果不满足这个条件，则执行第二条打印语句。由于要么满足条件，要么不满足条件，因此只有一个分支会被执行。

对于只有一条语句的分支来说，大括号是可选的，因此前一个示例可以写成下面这样：

```
if (x % 2 == 0)
    System.out.println("x is even");
else
    System.out.println("x is odd");
```

然而，即便大括号是可有可无的，最好不要省略，这样可避免在 if 或 else 代码块中添加语句时因忘记加大括号而导致错误。

```
if (x > 0)
    System.out.println("x is positive");
    System.out.println("x is not zero");
```

这些代码没有正确地缩进，因此极具误导性。由于省略了大括号，只有第一个 println 是 if 语句的一部分。在编译器看来，上述代码实际上是以下这样的：

```
if (x > 0) {
    System.out.println("x is positive");
```

```
}
    System.out.println("x is not zero");
```

因此，在任何情况下都将执行第二条 `println` 语句。即便是经验丰富的程序员也会犯这样的错误，只要在网上搜索 goto fail 就会有所了解了。

5.4　串接和嵌套

有时需要检查多个相关的条件，并在多种措施中选择一种方式。其中一种方法是将一系列的 `if` 和 `else` 语句*串接*（chaining）起来：

```
if (x > 0) {
    System.out.println("x is positive");
} else if (x < 0) {
    System.out.println("x is negative");
} else {
    System.out.println("x is zero");
}
```

你可以想串接多长就串接多长，但太长可能难以阅读。为提高可读性，可使用标准的缩进方式，如这里的示例所示。将语句和大括号对齐可降低出现语法错误的可能性。

除了串接，还可以在一个条件语句中*嵌套*（nesting）另一个条件语句以作出复杂的决策。可将前面的示例重写为下面这样：

```
if (x == 0) {
    System.out.println("x is zero");
} else {
    if (x > 0) {
        System.out.println("x is positive");
    } else {
        System.out.println("x is negative");
    }
}
```

外面的条件语句有两个分支，第一个分支包含一条打印语句，第二个分支包含另一个条件语句，该条件语句也有两个分支。这两个分支都是打印语句，但它们本来也可以是条件语句。

这样的嵌套结构很常见，但很难快速地阅读它们。因此我们必须使用正确的缩进，以便这种结构易于理解。

5.5　标志变量

要想存储 `true` 或 `false` 的值，需要使用 `boolean` 变量，而要创建 `boolean` 变量，可像下面这样做：

```
boolean flag;
flag = true;
boolean testResult = false;
```

第 1 行是变量声明，第 2 行是赋值语句，而第 3 行在声明变量的同时给它赋值。由于关系运算符的结果为 `boolean` 值，因此可将比较结果存储在一个变量中：

```
boolean evenFlag = (n % 2 == 0);    // n为偶数时为true
boolean positiveFlag = (x > 0);     // x为正数时为true
```

其中的括号并非必不可少，但可以让代码更容易理解。以这种方式定义的变量被称为标志（flag），因为它指出或“标志”着条件是否满足。

定义标志变量后，就可以在条件语句中使用了：

```
if (evenFlag) {
    System.out.println("n was even when I checked it");
}
```

注意，你无需这样书写：`if (evenFlag == true)`，因为 `evenFlag` 就是 `boolean` 值，可用于表示条件。同理，要检查标志是否为 `false`，可像下面这样做：

```
if (!evenFlag) {
    System.out.println("n was odd when I checked it");
}
```


5.6 return语句

`return` 语句允许还未到达末尾就终止方法。使用 `return` 语句的原因之一是检测到了错误条件：

```
public static void printLogarithm(double x) {
    if (x <= 0.0) {
        System.err.println("Error: x must be positive.");
        return;
    }
    double result = Math.log(x);
    System.out.println("The log of x is " + result);
}
```

这个示例定义了一个名为 `printLogarithm` 的方法，该方法将一个 `double` 值作为形参（名为 `x`），用于检查 `x` 是否小于或等于 0，如果是这样的，那么就显示一条错误消息，再用 `return` 退出方法。这样将立即返回到调用这个方法的地方，而不执行这个方法的后面代码。

这个示例使用了 `System.err`，这是一个 `OutputStream`，通常用于显示错误消息和警告。对于 `System.err` 的输出，有些开发环境用不同的颜色显示或在独立的窗口中显示。

5.7　验证输入

以下的方法调用了 5.6 节中的 printLogarithm：

```
public static void scanDouble() {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a number: ");
    double x = in.nextDouble();
    printLogarithm(x);
}
```

这个示例调用了 nextDouble，因此 Scanner 将尝试读取一个 double 值。如果用户输入的是一个浮点数，Scanner 将把它转换为 double 值；但如果用户输入的是其他类型的值，Scanner 将引发 InputMismatchException 异常。

为防范这种错误，可在分析前检查输入：

```
public static void scanDouble() {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a number: ");
    if (!in.hasNextDouble()) {
        String word = in.next();
        System.err.println(word + " is not a number");
        return;
    }
    double x = in.nextDouble();
    printLogarithm(x);
}
```

Scanner 类提供了方法 hasNextDouble，该方法用于检查能否将输入流中的下一个标记转换为 double 值。如果答案是肯定的，那么就可以调用 nextDouble，且不会引发异常。如果答案是否定的，那么就显示一条错误消息并返回。从 main 方法返回将导致程序终止。

5.8　递归方法

介绍了条件语句后，现在可以探索程序能做的最神奇的事情之一了——*递归*（recursion）。思考下面的示例：

```
public static void countdown(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        System.out.println(n);
        countdown(n - 1);
    }
}
```

这个方法名为 countdown，它将一个整数作为参数。如果这个参数为零，它就显示单词

Blastoff，否则就显示这个数字，再用实参 `n-1` 调用自己。调用自己的方法称为*递归方法*（recursive method）。

如果在 `main` 中调用 `countdown(3)`，结果将如何？

> 这次执行 countdown 时，n==3；由于 n 不为零，因此它显示值 3，再调用自己……
>
> > 这次执行 countdown 时，n==2；由于 n 不为零，因此它显示值 2，再调用自己……
> >
> > > 这次执行 countdown 时，n==1；由于 n 不为零，因此它显示值 1，再调用自己……
> > >
> > > > 这次执行 countdown 时，n==0；由于 n 为零，因此它显示值“Blastoff!”再返回。
> > >
> > > 使用 1 调用的 countdown 返回。
> >
> > 使用 2 调用的 countdown 返回。
>
> 使用 3 调用的 countdown 返回。

至此返回了 `main`，于是输出类似于下面这样：

```
3
2
1
Blastoff!
```

再来看一个示例。在这个示例中，我们将重新编写 4.3 节中的方法 `newLine` 和 `threeLine`：

```
public static void newLine() {
    System.out.println();
}

public static void threeLine() {
    newLine();
    newLine();
    newLine();
}
```

虽然这些方法管用，但如果要显示 2 个或 100 个空行，它们并不能提供帮助。下面是一种更佳的解决方案：

```
public static void nLines(int n) {
    if (n > 0) {
        System.out.println();
        nLines(n - 1);
    }
}
```

这个方法将一个整数（`n`）作为形参，并显示了 `n` 个空行。其结构与 `countdown` 类似：只要

n 大于零，就显示一个空行，然后调用自己再显示 *n*−1 个空行。显示的总空行数为 1+(`n` − 1)，即我们希望的 *n* 个。

5.9 递归栈图

第 4 章用栈图来表示程序在方法调用期间的状态，使用相同的图可让递归方法更容易解释。

前面说过，每当方法被调用时，Java 都会创建一个新的栈帧，其中包含该方法的形参和变量。图 5-1 是使用 3 调用 `countdown` 时的栈图。

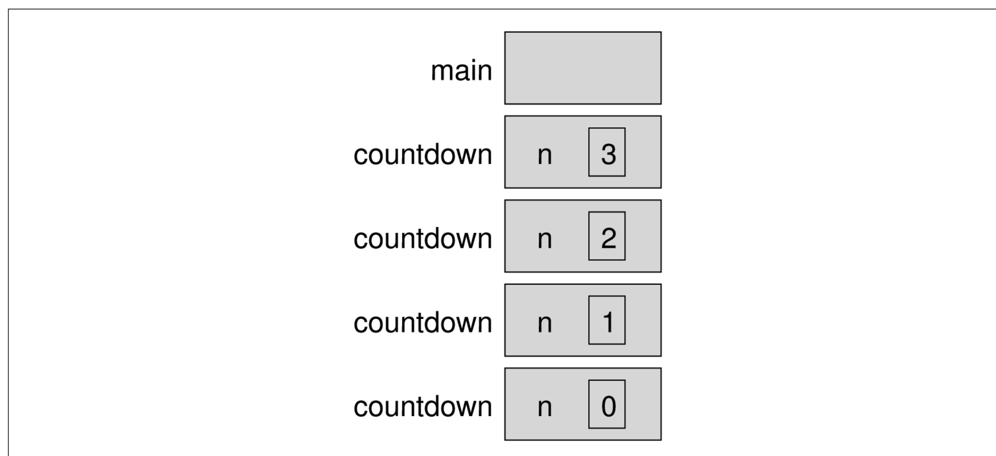


图 5-1：程序 `countdown` 的栈图

根据约定，`main` 的栈帧位于最上面，而栈是向下延伸的。`main` 的栈帧是空的，因为它不包含任何变量。（它包含形参 `args`，但由于没用这个形参，因此没有在栈图中列出。）

`countdown` 有四个栈帧，分别与形参 `n` 的不同值对应。在最后一个栈帧中 `n=0` 被称为基线条件（base case）。因为它没有执行递归调用，所以下面没有其他的栈帧。

如果递归方法不包含基线条件，或永远不能满足基线条件，那么栈将没完没了地增大，至少从理论上说如此。实际上，栈的长度是受限制的，超过限制将引发 `StackOverflowError` 异常。

例如，下面的递归方法就不包含基线条件：

```
public static void forever(String s) {
    System.out.println(s);
    forever(s);
}
```

这个方法将不断地显示指定的字符串，直到栈溢出，进而引发异常。

5.10 二进制数

前面的方法 countdown 由三部分组成：(1) 检查基线条件；(2) 显示输出；(3) 执行递归调用。如果将第 2 步和第 3 步的顺序调换一下，即先执行递归调用再显示输出，你认为结果将如何呢？

```
public static void countup(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        countup(n - 1);
        System.out.println(n);
    }
}
```

栈图与以前相同，并且这个方法也将被调用 *n* 次，但 System.out.println 将在递归调用返回前执行，因此结果是顺数而不是倒数：

```
Blastoff!
1
2
3
```

在按相反的顺序更容易计算结果时，可利用这种行为。例如，要想将十进制整数转换为用二进制（binary）表示，可反复地除以 2：

```
23 / 2 is 11 remainder 1
11 / 2 is  5 remainder 1
 5 / 2 is  2 remainder 1
 2 / 2 is  1 remainder 0
 1 / 2 is  0 remainder 1
```

从下往上阅读这些余数就可得到 23 的二进制表示——10111。要想了解更多有关二进制的详细信息，请参阅 http://www.mathsisfun.com/binary-number-system.html。

下面的递归方法可用于显示任何正整数的二进制表示：

```
public static void displayBinary(int value) {
    if (value > 0) {
        displayBinary(value / 2);
        System.out.print(value % 2);
    }
}
```

如果 value 为 0，displayBinary 就什么都不做（这就是基线条件）。如果传入的实参为正，该方法就将其除以 2，再递归地调用自己。这个方法在递归调用返回前显示结果中的一位。

最左边的那位位于栈底，因此最先显示；最右边的那位位于栈顶，因此最后显示。调用 displayBinary 后，我们可以用 println 指出输出到此结束：

```
displayBinary(23);
System.out.println();
// 输出为10111
```

要想像计算机科学家那样思考，学会递归思维很重要。向上或向下执行计算的递归算法可以用简洁的方式实现很多算法。

5.11 术语表

- boolean

  一种只有两个可能取值（true 和 false）的数据类型。

- 关系运算符

  对两个值进行比较，从而得到一个表示两者关系的 boolean 值的运算符。

- 逻辑运算符

  将两个 boolean 值合并成一个 boolean 值的运算符。

- 短路

  执行逻辑运算符的一种方式，仅在必要时计算第二个操作数的值。

- 德·摩根定律

  指出如何对逻辑表达式求反的数学规则。

- 条件语句

  根据条件确定执行哪些语句的语句。

- 分支

  条件语句中多个可能执行的语句块之一。

- 串接

  一种将多个条件语句依次连接起来的方式。

- 嵌套

  在一个条件语句的分支中包含另一个条件语句。

- 标志

  表示条件或状态的变量，通常为 boolean 变量。

- *递归*

  在当前执行的方法中再调用该方法的过程。

- *递归方法*

  调用自己的方法，通常提供不同的实参。

- *基线条件*

  导致递归方法不再进行递归调用的条件。

- *二进制*

  只用 0 和 1 表示数字的计数系统，也被称为“以 2 为基数”的计数系统。

5.12 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch05 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

如果你还没有阅读 A.6 节，那么现在正是阅读的好时机。该节介绍了 DrJava 调试器，这是一个很有用的执行流程跟踪工具。

练习5-1

逻辑运算符可简化嵌套的条件语句。例如，你能只用一条 `if` 语句重写下面的代码吗？

```
if (x > 0) {
    if (x < 10) {
        System.out.println("positive single digit number.");
    }
}
```

练习5-2

根据下面的程序：

(1) 绘制一个栈图，指出*第二次*调用 `ping` 时该程序的状态；

(2) 这个程序的完整输出是什么？

```
public static void zoop(String fred, int bob) {
    System.out.println(fred);
    if (bob == 5) {
        ping("not ");
    } else {
        System.out.println("!");
    }
}

public static void main(String[] args) {
```

```
        int bizz = 5;
        int buzz = 2;
        zoop("just for", bizz);
        clink(2 * buzz);
    }

    public static void clink(int fork) {
        System.out.print("It's ");
        zoop("breakfast ", fork) ;
    }

    public static void ping(String strangStrung) {
        System.out.println("any " + strangStrung + "more ");
    }
```

练习5-3

对于 5.8 节中的程序，假设在 `main` 中用实参 4 调用了方法 `nLines`，请据此绘制一个栈图，并指出 `nLines` 的最后一次调用返回前，该程序的状态如何。

练习5-4

费马大定理指出，如果整数 n 大于 2，则方程 $a^n+b^n=c^n$ 没有整数解。

请编写一个名为 `checkFermat` 的方法，它接受四个整数作为参数：`a`、`b`、`c` 和 `n`，然后检查费马大定理是否正确。如果 n 大于 2，且 $a^n+b^n=c^n$，那么这个方法就显示“Holy smokes, Fermat was wrong!”，否则就显示“No, that doesn’t work.”。

提示：你可能需要使用 `Math.pow`。

练习5-5

这个练习的目的是将一个大问题分解成多个小问题，并编写简单的方法来解决这些小问题。下面是歌曲 *99 Bottles of Beer* 的第一段歌词：

> 99 bottles of beer on the wall,
> 99 bottles of beer,
> ya’ take one down, ya’ pass it around,
> 98 bottles of beer on the wall.

接下来的各段歌词与此相同，只是每段中的瓶数依次少 1。而最后一段歌词如下：

> No bottles of beer on the wall,
> no bottles of beer,
> ya’ can’t take one down, ya’ can’t pass it around,
> ’cause there are no more bottles of beer on the wall!

至此，这个歌曲总算结束了。

请编写一个显示这首歌全部歌词的程序。程序应包含一个承担重任的递归方法，但你可能还需要编写其他方法。可在程序编写期间用较少的段数（如 3）进行测试。

练习5-6

这个练习将通过一个包含多个方法的程序来复习执行流程。请阅读下面的代码，并回答后面的问题。

```
public class Buzz {

    public static void baffle(String blimp) {
        System.out.println(blimp);
        zippo("ping", -5);
    }

    public static void zippo(String quince, int flag) {
        if (flag < 0) {
            System.out.println(quince + " zoop");
        } else {
            System.out.println("ik");
            baffle(quince);
            System.out.println("boo-wa-ha-ha");
        }
    }

    public static void main(String[] args) {
        zippo("rattle", 13);
    }

}
```

(1) 在这个程序首先执行的代码行旁边标上数字 1。

(2) 在这个程序执行的第 2 行代码旁边标上数字 2，再依此类推，直到该程序末尾。多次执行的代码行旁边将有多个数字。

(3) 调用 `baffle` 时，形参 `blimp` 的值是什么？

(4) 这个程序的输出是什么？

练习5-7

学习条件语句后，现在我们可以回过头去完善练习 3-4 中的“猜数”游戏了。

你应该已经有这样一个程序，它可以选择一个随机数，让用户猜这个数，并显示用户猜的数字与这个随机数的差距。

请修改这个程序，让程序可以告诉用户他猜的数字太大还是太小，并提示用户再猜一次。每次修改只添加少量的代码，并边修改边测试。

这个程序应不断运行，直到用户猜对为止。

提示：使用两个方法，并将其中一个编写为递归方法。

第 6 章

值方法

在本书的前几章中，我们使用的一些方法会返回值，如 Math 类的方法。但我们编写的方法都是 void 方法，即不返回值。我们将在本章编写返回值的方法，这些方法被称为*值方法*（value method）。

6.1 返回值

调用 void 方法时，调用代码通常独占一行。例如，以下是 5.8 节中的 countup 方法：

```
public static void countup(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        countup(n - 1);
        System.out.println(n);
    }
}
```

下面的代码演示了如何调用它：

```
countup(3);
System.out.println("Have a nice day.");
```

另一方面，调用值方法时，必须对其返回值做些什么。我们通常将其赋给变量或用于表达式中，如下所示：

```
double error = Math.abs(expected - actual);
double height = radius * Math.sin(angle);
```

与 void 方法相比，值方法有两个不同之处：

- 声明了返回值的类型，即*返回类型*（return type）；
- 至少使用了一条 return 语句来提供*返回值*（return value）。

下面是一个示例：calculateArea 接受一个 double 参数，并返回以该参数值为半径的圆的面积：

```
public static double calculateArea(double radius) {
    double result = Math.PI * radius * radius;
    return result;
}
```

与本书前面一样，这个方法也是公有的、静态的，但在以前为 void 的地方使用了 double，这意味着该方法的返回值为 double 值。

最后一行是一种新型的 return 语句，其中包含一个返回值。这条语句的意思是，立即从这个方法返回，并将接下来的表达式用作返回值。这个表达式的复杂程度没有限制，因此我们可以更简洁地编写这个方法：

```
public static double calculateArea(double radius) {
    return Math.PI * radius * radius;
}
```

然而，使用 result 这样的*临时变量*（temporary variable）通常可以简化调试工作，在用交互式调试器（参见 A.6 节）以步进方式执行代码时尤其如此。

return 语句中的表达式类型必须与方法的返回类型一致。将返回类型声明为 double 时，其实就是承诺这个方法最终将生成一个 double 值；如果使用的 return 语句不包含表达式或包含的表达式的类型不正确，那么编译器将显示错误消息。

有时候，使用多条 return 语句很有用，例如，在条件语句的每个分支中使用一条：

```
public static double absoluteValue(double x) {
    if (x < 0) {
        return -x;
    } else {
       return x;
    }
}
```

由于这些 return 语句位于条件语句中，因此只会执行其中一条。只要执行了其中任何一条 return 语句，这个方法就会立即结束，不再执行其他的语句。

位于 return 语句后，且与之同属一个代码块的代码以及其他地方根本不会执行的代码被称为*无用代码*（dead code）。如果程序包含无用代码，编译器将显示错误消息——unreachable

statement（不可达语句）。例如，下面的方法就包含无用代码：

```
public static double absoluteValue(double x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
    System.out.println("This line is dead.");
}
```

将 `return` 语句放在条件语句中时，必须确保程序中的每条可能的执行路径都包含一条 `return` 语句。如果情况并非如此，编译器将指出这一点。例如，下面的方法就不完整：

```
public static double absoluteValue(double x) {
    if (x < 0) {
        return -x;
    } else if (x > 0) {
        return x;
    }
    // 语法错误
}
```

如果 `x` 为 0，那么将不满足任何上述条件，这会导致方法的执行路径中没有 `return` 语句。在这种情况下，编译器显示的错误消息可能类似于这样：missing return statement（缺少 `return` 语句）。这条错误消息令人迷惑，因为已经有两条 `return` 语句了。希望你以后遇到这种情况时知道这是怎么回事。

6.2 编写方法

初学者常常犯这样的错误，即在不尝试编译并运行的情况下就编写大量代码，然后再花大量的时间调试代码。我们认为渐进开发（incremental development）是一种更好的方法，这种方法的要点如下。

- 先编写一个能够运行的简单程序，再逐步修改。这样的话，无论什么时候出现错误，你都知道该检查哪些地方。
- 用变量存储中间值，以便能够用打印语句或调试器检查它们。
- 程序能够正确运行后再将多条语句合并为复合表达式（前提是这样不会导致程序更难理解）。

举个例子，假设你要根据两个点的坐标 (x_1, y_1) 和 (x_2, y_2) 计算它们之间的距离，可使用下面的公式：

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

首先要考虑的是，用 Java 编写 `distance` 方法会是什么样的呢？换而言之，输入（形参）是什么？输出（返回值）又是什么？在这个示例中，输入是两个点；而要表示这两个点，显而易见的方法是使用 4 个 `double` 值。返回值是距离，其类型也应为 `double`。

据此就可以编写这个方法的轮廓了，这种轮廓有时被称为*存根*（stub），其中包含了方法的特征标和一条 `return` 语句：

```
public static double distance
        (double x1, double y1, double x2, double y2) {
    return 0.0;
}
```

这条 `return` 语句是一个占位符，在确保程序能够通过编译方面必不可少。当前，这个程序并没有做什么有用的事情，但添加更多代码前有必要对其进行编译，以便发现语法错误。

开发新方法前先考虑测试是一个不错的主意，能够帮助你搞明白如何实现这些方法。要测试方法，可在 `main` 中调用，并将样本值用作实参：

```
double dist = distance(1.0, 2.0, 4.0, 6.0);
```

坐标指定的两点间的水平距离为 3.0，垂直距离为 4.0，因此结果应为 5.0。测试方法时知道正确答案很有帮助。

存根通过编译后，我们就可以开始添加代码了——每次一行。每次修改后都再次编译并运行程序。无论什么时候出现错误，我们都很清楚该检查什么地方：添加的最后一行代码。

下一步是计算（$x_2 - x_1$）和（$y_2 - y_1$）的差值。我们将这些值分别存储在临时变量 `dx` 和 `dy` 中。

```
public static double distance
        (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    System.out.println("dx is " + dx);
    System.out.println("dy is " + dy);
    return 0.0;
}
```

其中的打印语句让我们能够检查这些中间值，它们应该分别为 3.0 和 4.0，确定它们正确后再继续添加代码。可在方法编写好后删除这些打印语句；类似这样的代码被称为*脚手架*（scaffolding），因为它们可以帮助你创建程序，却不是最终产品的组成部分。

下一步是计算 `dx` 和 `dy` 的平方和。为此可使用方法 `Math.pow`，但更简单的方式是将它们分别与自己相乘：

```
public static double distance
        (double x1, double y1, double x2, double y2) {
```

```
        double dx = x2 - x1;
        double dy = y2 - y1;
        double dsquared = dx * dx + dy * dy;
        System.out.println("dsquared is " + dsquared);
        return 0.0;
    }
```

同样，此时应编译并运行该程序，同时也要检查中间值——应为 25.0。终于，我们可以用 Math.sqrt 来计算并返回结果：

```
    public static double distance
            (double x1, double y1, double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        double dsquared = dx * dx + dy * dy;
        double result = Math.sqrt(dsquared);
        return result;
    }
```

等编程经验更丰富后，你就可以一次编写并调试多行代码。即便如此，渐进开发也可为你节省大量的时间。

6.3　方法组合

定义新方法后，可将其用于表达式中或用来创建其他方法。例如，假设有人向你提供了两个点——圆心和圆周上的一点，并要求你计算这个圆的面积。那么我们可以假设圆心坐标存储在变量 xc 和 yc 中，而圆周上那个点的坐标存储在 xp 和 yp 中。

第一步要做的是计算出这个圆的半径，即两个点间的距离。所幸我们已经有执行这种计算的方法——distance：

```
    double radius = distance(xc, yc, xp, yp);
```

第二步是根据得到的半径计算圆的面积。我们也有执行这种计算的方法——calculateArea：

```
    double area = calculateArea(radius);
    return area;
```

将这些代码放到一个新的方法中，结果如下：

```
    public static double circleArea
            (double xc, double yc, double xp, double yp) {
        double radius = distance(xc, yc, xp, yp);
        double area = calculateArea(radius);
        return area;
    }
```

临时变量 radius 和 area 对开发和调试来说很有用，但是程序能正确运行后就可以组合方法调用，从而让程序更简洁：

```
public static double circleArea
        (double xc, double yc, double xp, double yp) {
    return calculateArea(distance(xc, yc, xp, yp));
}
```

这个示例演示了*功能分解*（functional decomposition）的过程，即将复杂的计算分成简单的方法，分别对这些方法进行测试，然后再组合这些方法来执行计算。这种做法可缩短调试时间，并且编写出来的代码准确度更高且更容易维护。

6.4 重载

你可能注意到了，circleArea 和 calculateArea 的功能类似，都用于计算圆的面积，只是接受的参数不同：要想调用 calculateArea，必须提供半径，而调用 circleArea 则需要提供两个点的坐标。

如果两个方法所做的事情相同，那么自然应该给它们指定相同的名称。多个方法同名被称为*重载*（overloading），这在 Java 中是合法的，但前提条件是每个版本接受的参数不同。因此，我们可以将 circleArea 重命名为 calculateArea：

```
public static double calculateArea
        (double xc, double yc, double xp, double yp) {
    return calculateArea(distance(xc, yc, xp, yp));
}
```

请注意，这个方法并不是递归方法。当调用重载的方法时，Java 会根据你提供的实参判断要调用的是哪个版本。如果编写如下代码：

```
double x = calculateArea(3.0);
```

Java 将查找一个以 double 值作为参数的方法 calculateArea，因此它将使用第一个版本，这个版本将提供的实参视为半径。如果编写如下代码：

```
double y = calculateArea(1.0, 2.0, 4.0, 6.0);
```

Java 将使用 calculateArea 的第二个版本，将提供的实参视为两个点的坐标。在这个示例中，第二个版本调用了第一个版本。

很多 Java 方法都是重载的，这意味着它们有不同的版本，但每个版本的形参类型或数量各不相同。例如，print 和 println 都有接受单个参数（任何数据类型均可）的版本；而 Math 类的 abs 方法既有用于 doube 值的版本，也有用于 int 值的版本。

虽然重载是一项很有用的功能，但还是应该慎用。如果在调试方法的一个版本时，不小心

调用了另一个版本，你可能会把自己绕进去。

6.5 boolean方法

方法可返回任何类型的值，当然也包括 boolean 值。返回 boolean 值的方法非常适合隐藏测试，如下所示：

```
public static boolean isSingleDigit(int x) {
    if (x > -10 && x < 10) {
        return true;
    } else {
        return false;
    }
}
```

这个方法名为 isSingleDigit。通常会给 boolean 方法指定一个像一般疑问句的名称。因为返回类型为 boolean，所以 return 语句中的表达式必须是 boolean 表达式。

虽然这些代码比实际需要的要长，但是很简单。表达式 x >- 10 && x < 10 的类型就是 boolean，因此完全可以直接返回它（不需要 if 语句）：

```
public static boolean isSingleDigit(int x) {
    return x > -10 && x < 10;
}
```

在 main 中，你可以像通常那样调用这个方法：

```
System.out.println(isSingleDigit(2));
boolean bigFlag = !isSingleDigit(17);
```

第 1 行显示 true，因为 2 是一个个位数；第 2 行将 bigFlag 设置为 true，因为 17 不是个位数。

条件语句常常将 boolean 方法的结果用作条件：

```
if (isSingleDigit(z)) {
    System.out.println("z is small");
} else {
    System.out.println("z is big");
}
```

像这样的语句几乎都可解读为“如果 z 是个位数，就打印……否则打印……”。

6.6 Javadoc标签

我们在 4.9 节中讨论了如何用 /** 编写文档注释。一般而言，最好给每个类和方法都编写

文档，这样其他程序员无需阅读代码就能知道它们是做什么的。

为将文档分成多个部分，Javadoc 支持以 @ 打头的可选标签（tag）。例如，可用标签 `@param` 和 `@return` 提供有关形参和返回值的额外信息。

```
/**
 * 检查整数x是否是个位数
 *
 * @param x 要检查的整数
 * @return 如果x是个位数,返回true,否则返回false
 */
public static boolean isSingleDigit(int x) {
```

图 6-1 显示了 Javadoc 生成的 HTML 页面。请注意其中的源代码和文档之间的关系。

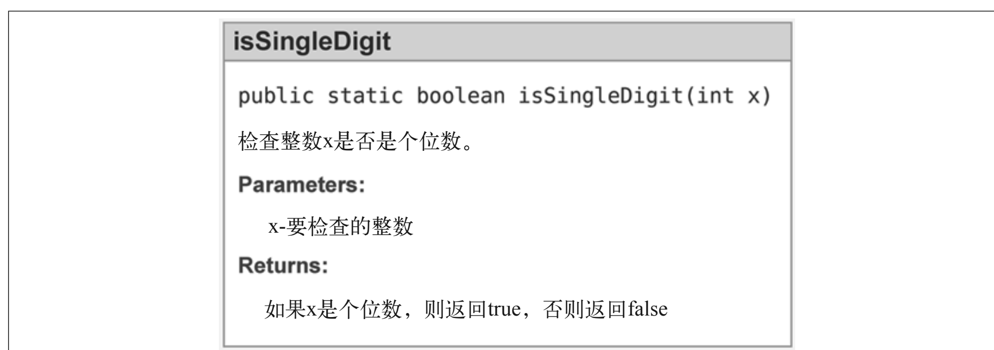


图 6-1：`isSingleDigit` 的 HTML 文档

如果方法有多个形参，那么应该用不同的 `@param` 标签描述每个形参。`void` 方法没有返回值，因此不需要 `@return` 标签。

6.7　再谈递归

学习完返回值的方法后，你掌握的 Java 编程知识就是*图灵完备的*（Turing complete）了。这意味着你能用 Java 来计算任何可计算的东西，只要"可计算"的定义是合理的。这个概念是由阿隆佐 · 邱奇（Alonzo Church）和阿兰 · 图灵（Alan Turing）提出的，因此被称为邱奇–图灵论题。

为了让你了解学过的工具能做哪些事情，我们来看一些方法，这些方法可用于计算以递归方式定义的函数。递归定义引用了当前定义的东西，从这种意义上说，它类似于循环定义。

当然，真正意义上的循环定义用处不大，请看以下定义。

- *递归*（recursive）
  用于描述递归的方法。

如果在字典中看到这个定义，你可能非常恼火。但如果用 Google 搜索 recursion，它将显示 Did you mean: recursion，这是一个只有知道内情的人才懂的玩笑。

很多数学函数都是以递归的方式定义的，因为这常常是最简单的方式。例如，整数 n 的阶乘（factorial，表示为 $n!$）的定义类似于以下这样：

$$0! = 1$$

$$n! = n \cdot (n - 1)!$$

请不要将数学符号 ! 同 Java 运算符 ! 混为一谈，前者表示阶乘，后者表示逻辑非。上述定义指出，`factorial(0)` 为 `1`，而 `factorial(n)` 为 `n*factorial(n - 1)`。

因此，`factorial(3)` 为 `3*factorial(2)`；`factorial(2)` 为 `2*factorial(1)`；`factorial(1)` 为 `1*factorial(0)`；`factorial(0)` 为 `1`。通过整合，可以得到 `factorial(3)` 为 `3*2*1*1`，即 6。

如果能用公式表示递归定义，那么就可以轻松地编写有关的计算方法。为此，首先需要确定形参和返回类型。因为阶乘是针对整数定义的，所以计算阶乘的方法接受一个 `int` 形参，并返回一个 `int` 值。以下代码很好地表示了这一点：

```
public static int factorial(int n) {
    return 0;
}
```

接下来需要考虑基线条件。如果提供的实参为 0，则返回 1。

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return 0;
}
```

否则，就需要执行递归调用来计算 n-1 的阶乘，并将结果乘以 n（这是比较有趣的部分）：

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    int recurse = factorial(n - 1);
    int result = n * recurse;
    return result;
}
```

这个程序的执行流程与 5.8 节中的 `countdown` 类似。如果我们用 3 来调用 `factorial`，执行流程将如下所示。

因为 n 为 3，所以执行第二个分支——计算 n–1 的阶乘……

　　因为 n 为 2，所以执行第二个分支——计算 n–1 的阶乘……

　　　　因为 n 为 1，所以执行第二个分支——计算 n–1 的阶乘……

　　　　　　因为 n 为 0，所以执行第一个分支——立即返回 1。

　　　　将返回值（1）乘以 n（1），并将结果返回。

　　将返回值（1）乘以 n（2），并将结果返回。

将返回值（2）乘以 n（3），并将结果（6）返回给 `factorial(3)` 的调用者。

图 6-2 显示了这个调用序列的栈图，其中的返回值沿栈向上传递。注意，最后一个栈帧中没有 `recurse` 和 `result`，这是因为当 `n == 0` 时，声明了这些变量的代码不会被执行。

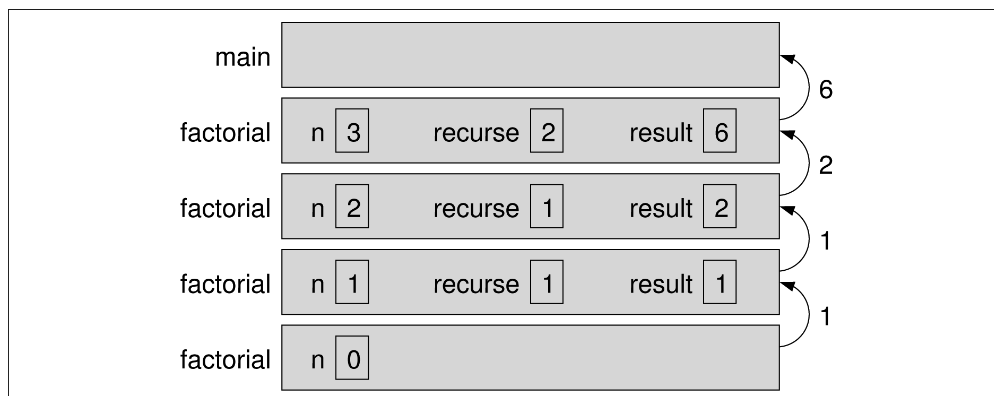


图 6-2：方法 `factorial` 的栈图

6.8　姑且相信

遵循执行流程是阅读程序的方式之一，但这种方式很快就会让你不堪重负。另一种方式是**姑且相信**（leap of faith）：遇到方法调用时，不沿执行流程前行，而假设被调用的方法能够正确地工作并返回合适的值。

事实上，你在用 Java 库中的方法时，就践行了姑且相信的理念：调用 `Math.cos` 或 `System.out.println` 时，你并不检查这些方法的实现，而是假定它们能够正确地工作。

对于自己编写的方法，你也应该采取这种做法。例如，我们在 6.5 节中编写了一个名为 `isSingleDigit` 的方法，用来判断一个数字是否在 0~9。通过测试和检查代码确定这个方法正确后，我们就可在使用时免去重复查看实现。

递归方法也应如此。遇到递归调用时，应假定递归调用可以正确工作，而不沿执行流程前行。例如，“如果能计算出 n–1 的阶乘，就能计算出 n 的阶乘，这样对吗？”没错，只需

再乘以 n 即可。

当然，假定未编写好的方法能够正确地运行让人觉得很别扭，这正是我们称之为“姑且相信”的原因所在！

6.9 再举一个例子

另一个以递归方式定义的常见函数是斐波纳契数列，其定义如下：

$$fibonacci(1) = 1$$

$$fibonacci(2) = 1$$

$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$$

如果将该定义转换为 Java 代码，结果如下：

```
public static int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

如果试图沿着上述代码的流程执行，即便 `n` 的值很小，你也会遇到很多麻烦。但如果采取姑且相信的做法，即假设两个递归调用能够正确地工作，那么就能清楚地知道结果就是这两个递归调用的返回值之和。

6.10 术语表

- void方法

  不返回值的方法。

- 值方法

  返回值的方法。

- 返回类型

  方法返回的值的类型。

- 返回值

  方法调用的结果。

- 临时变量

  短期的变量，通常用于调试。

- *无用代码*

  程序中根本不会执行的代码，通常是因为其位于 `return` 语句的后面。

- *渐进开发*

  创建程序的一种流程，每次只编写几行代码，并立即进行编译和测试。

- *存根*

  未完成的方法的占位符，旨在让类能够通过编译。

- *脚手架*

  在程序开发期间使用但不包含在最终版本中的代码。

- *功能分解*

  将复杂的计算分成多个简单的方法，再组合这些方法来执行计算。

- *重载*

  定义了多个名称相同但形参不同的方法。

- *标签*

  以 `@` 打头的标记，Javadoc 根据它们将文档分成多个部分。

- *图灵完备的*

  编程语言能够实现任何理论上可行的算法。

- *阶乘*

  将从 1 到给定整数的所有整数相乘。

- *姑且相信*

  阅读递归程序的一种方式，假设递归调用可以正确地工作，而不沿执行流程前行。

6.11 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch06 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

如果你还没有阅读 A.7 节，那么现在正是阅读的好时机。该节介绍了 JUnit——一款测试值方法的有效工具。

练习6-1

如果你对某种做法是否合法、不合法的结果是什么存在疑问，让编译器来解答是一种不错的方法。请尝试回答下面的问题。

(1) 如果调用一个值方法，但不用它返回的结果，即既没有将其赋给变量，也没有将其用于表达式中，结果将如何呢？
(2) 如果将一个 `void` 方法用于表达式中，结果将如何呢？例如，尝试执行代码 `System.out.println("boo!") + 7`。

练习6-2

编写一个名为 `isDivisible` 的方法，让它接受两个整数——`n` 和 `m`，并在 `n` 能被 `m` 整除时返回 `true`，否则返回 `false`。

练习6-3

给定 3 根棍子，可能能够将它们排列成三角形，也可能不行。例如，如果其中一根棍子长 12 英寸，另外两根都为 1 英寸，那么就无法将 3 根棍子相互相连。给定任意 3 根棍子的长度，可通过下面的简单测试来判断它们能否组成三角形：

> 只要其中 1 根棍子的长度大于其他 2 根棍子的总长，它们就无法组成三角形。

请编写一个名为 `isTriangle` 的方法，让它接受 3 个整数参数，并根据长度分别为这 3 个整数的棍子能否组成三角形而返回 `true` 或 `false`。这个练习的重点是用条件语句编写值方法。

练习6-4

很多计算都可用"乘加"运算来更简洁地表示出来，这种运算接受 3 个操作数，并计算 `a * b + c` 的结果。有些处理器甚至提供了对浮点数执行这种运算的硬件实现。

(1) 创建一个程序，并命名为 Multadd.java。
(2) 编写一个名为 `multadd` 的方法，让它接受 3 个 `double` 参数，并返回 `a * b + c`。
(3) 编写一个 `main` 方法，并在其中测试方法 `multadd`：调用 `multadd` 并传递几个简单的实参，如 `1.0`、`2.0`、`3.0`。
(4) 另外，在 `main` 中用方法 `multadd` 来计算下述表达式的值：

$$\sin\frac{\pi}{4}+\frac{\cos\frac{\pi}{4}}{2}$$

$$\log 10+\log 20$$

(5) 编写一个名为 `expSum` 的方法，让它接受一个 `double` 参数，并调用 `multadd` 来计算下述表达式的值：

$$xe^{-x}+\sqrt{1-e^{-x}}$$

在这个练习的最后一部分，你需要编写一个方法，这个方法要能调用你编写的另一个方法。在这种情况下，最好先仔细测试要调用的方法，否则你可能面临同时调试两个方法的

困境。

这个练习的目的之一是锻炼你的模式匹配能力——能够看出要解决的问题是通用问题的特例。

练习6-5

以下程序的输出是什么？

```
public static void main(String[] args) {
    boolean flag1 = isHoopy(202);
    boolean flag2 = isFrabjuous(202);
    System.out.println(flag1);
    System.out.println(flag2);
    if (flag1 && flag2) {
        System.out.println("ping!");
    }
    if (flag1 || flag2) {
        System.out.println("pong!");
    }
}

public static boolean isHoopy(int x) {
    boolean hoopyFlag;
    if (x % 2 == 0) {
        hoopyFlag = true;
    } else {
        hoopyFlag = false;
    }
    return hoopyFlag;
}

public static boolean isFrabjuous(int x) {
    boolean frabjuousFlag;
    if (x > 0) {
        frabjuousFlag = true;
    } else {
        frabjuousFlag = false;
    }
    return frabjuousFlag;
}
```

这个练习旨在确保你明白逻辑运算符和值方法的执行流程。

练习6-6

在这个练习中，你可以用栈图来帮助理解下述递归程序的执行流程：

```
public static void main(String[] args) {
    System.out.println(prod(1, 4));
}

public static int prod(int m, int n) {
    if (m == n) {
```

```
            return n;
        } else {
            int recurse = prod(m, n - 1);
            int result = n * recurse;
            return result;
        }
    }
```

(1) 绘制一个栈图，指出对 prod 的最后一次调用结束前，这个程序的状态是什么样的。
(2) 这个程序的输出是什么？（先尝试用纸和笔来回答这个问题，再通过运行代码来检查答案。）
(3) 简单地说说 prod 是做什么的（无需涉及其工作原理的细节）。
(4) 修改方法 prod，删除其中的临时变量 recurse 和 result。提示：else 分支只需要一行代码。

练习6-7

编写一个名为 oddSum 的递归方法，它接受一个正奇数——n，并返回 1~n（闭区间）所有奇数的和。请先考虑基线条件，并用临时变量调试解决方案。每次调用 oddSum 时都打印参数值 n 可能大有帮助。

练习6-8

这个练习的目标是将递归定义转换为 Java 方法。阿克曼函数的定义如下（其中 m 和 n 都是非负整数）：

$$A(m,n)=\begin{cases} n+1 & \text{如果} m=0 \\ A(m-1,1) & \text{如果} m \text{且} n=0 \\ A(m-1,A(m,n-1) & \text{如果} m>0 \text{且} n>0 \end{cases}$$

请编写一个名为 ack 的方法，让它接受两个 int 参数，然后计算并返回阿克曼函数的值。

测试你编写的 ack 方法：在 main 中调用它，并显示它返回的值。请注意，阿克曼函数值的递增速度非常快，测试时应使用较小的 m 和 n（不超过 3）。

练习6-9

编写一个名为 power 的递归方法，它接受 double 参数 x 和 int 参数 n，并返回 x^n 的值。

提示：这种运算的递归定义为 $x^n = x \cdot x^{n-1}$。另外别忘了，零以外的任何数字的 0 次方都为 1。

选做题：n 为偶数时，利用公式 $x^n=(x^{n/2})^2$ 来提高这个方法的效率。

第 7 章

循环

计算机常用于自动完成重复的任务。重复执行任务而不出错是计算机的强项，也是人类的弱项。

多次运行相同的代码被称为*迭代*（iteration）。我们在本书前面已经见过用递归来迭代的方法，如 `countdown` 和 `factorial`。虽然递归既优雅又强大，但需要使用一段时间才能习惯。Java 提供的语言功能简化了迭代，这些语言功能指的是 `while` 语句和 `for` 语句。

7.1 while语句

可用 `while` 语句改写本书前面的 `countdown` 方法：

```
public static void countdown(int n) {
    while (n > 0) {
        System.out.println(n);
        n = n - 1;
    }
    System.out.println("Blastoff!");
}
```

`while` 语句平白如话：只要 `n` 大于零，就打印 `n` 的值并将它减 1；`n` 为零后，打印“Blastoff!”。

括号内的表达式称为条件；而大括号内的语句称为*循环体*（loop body）。`while` 语句的执行流程如下。

(1) 计算条件，结果为 `true` 或 `false`。
(2) 如果条件为 `false`，就跳过循环体，接着执行循环体后面的语句。
(3) 如果条件为 `true`，就执行循环体，再回到第 1 步。

这种流程被称为*循环*（loop），因为执行完最后一步还会再回到第一步。

循环体应修改一个或多个变量的值，让条件最终为 `false`，从而结束循环。否则循环将没完没了地执行，这样的循环被称为*无限循环*（infinite loop）。计算机科学家总喜欢拿洗发水说明开涮，说其中的"抹洗发水、冲掉、再来"就是无限循环。

就前面的 `countdown` 方法而言，我们能够证明其中的循环肯定会结束。但一般而言，循环会不会结束并不那么容易判断。例如，下面的循环就会不断执行，直到 `n` 为 1（导致条件为 `false`）：

```
public static void sequence(int n) {
    while (n != 1) {
        System.out.println(n);
        if (n % 2 == 0) {          // n为偶数
            n = n / 2;
        } else {                   // n为奇数
            n = n * 3 + 1;
        }
    }
}
```

每次执行循环时都会显示 `n` 的值，然后再检查它是奇数还是偶数。如果是偶数，就除以 2；如果是奇数，就将它设置为 $3n+1$。例如，如果起始值（传递给 `sequence` 的实参）为 3，将生成数列 3、10、5、16、8、4、2、1。

因为 `n` 时而增大时而减小，所以没有明显的证据可以证明 `n` 终将变成 1，从而导致这个程序终止。`n` 为某些值时，我们能够证明这个程序终将终止。例如，如果 `n` 的起始值为 2 的幂，则每次执行循环时，`n` 都将为偶数，并最终会变成 1。在前面的数列中，从 `n` 为 16 开始就是一个这样的数列。

不管 `n` 的初始值是多少，这个程序最终都将结束吗？这是一个难以回答的问题。到目前为止，既没人能够证明这一点，也没人能够证伪！更多详细信息请参阅 https://en.wikipedia.org/wiki/Collatz_conjecture。

7.2 生成表格

循环非常适合用于生成和显示表格型数据。计算机还未面世时，人们必须手工计算对数、正弦、余弦等常见的数学函数。为简化这种工作，有些书提供了表格，让你能够查找各种函数的值，但手工创建这样的表格既缓慢又繁琐，并且结果常常错误百出。

计算机面世后，大家最初的反应之一是：太好了，可用计算机来生成这样的表格了，并确

保它们没有任何错误。事实证明确实如此，但人们的目光还是太短浅了，不久后，计算机就非常普及，这些打印出来的表格也就被淘汰了。

但对于有些运算来说，计算机依然要先在值表中查找近似结果，再通过计算来提高结果的精度。然而，有些值表存在错误，其中最著名的是最初的 Intel 奔腾处理器用来计算浮点数除法的值表（参见 https://en.wikipedia.org/wiki/Pentium_FDIV_bug）。

虽然“对数表”不再像以前那么有用了，但它依然是迭代的典范。下面的循环显示了一个表格，其中左边一列是一系列值，而右边一列是这些值的对数：

```
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + " " + Math.log(x));
    i = i + 1;
}
```

这个程序的输出如下：

```
1.0   0.0
2.0   0.6931471805599453
3.0   1.0986122886681098
4.0   1.3862943611198906
5.0   1.6094379124341003
6.0   1.791759469228055
7.0   1.9459101490553132
8.0   2.0794415416798357
9.0   2.1972245773362196
```

`Math.log` 计算自然对数，即以 e 为底的对数。在计算机应用领域常常需要计算以 2 为底的对数，为此可使用下面的公式：

$$\log_2 x = \frac{\log_e x}{\log_e 2}$$

我们可将前面的循环修改成下面这样：

```
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + " " + Math.log(x) / Math.log(2));
    i = i + 1;
}
```

结果如下：

```
1.0   0.0
2.0   1.0
3.0   1.5849625007211563
```

```
4.0   2.0
5.0   2.321928094887362
6.0   2.584962500721156
7.0   2.807354922057604
8.0   3.0
9.0   3.1699250014423126
```

我们在每次循环中都将 x 的值加 1，从而得到一个等差数列。如果不将 x 加 1，而是乘以一个常数，将得到一个等比数列：

```
final double LOG2 = Math.log(2);
int i = 1;
while (i < 100) {
    double x = (double) i;
    System.out.println(x + " " + Math.log(x) / LOG2);
    i = i * 2;
}
```

第 1 行将 Math.log(2) 存储在一个 final 变量中，以免反复计算这个表达式的值；最后一行将 x 乘以 2。结果如下：

```
1.0    0.0
2.0    1.0
4.0    2.0
8.0    3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

这个表格列出了 2 的幂及其以 2 为底的对数。虽然对数表已毫无用处，但对计算机科学家来说，知道 2 的幂大有裨益！

7.3　封装和泛化

6.2 节介绍了一种名为渐进开发的程序编写方法，本节介绍另一种*程序开发*（program development）流程——封装和泛化，步骤如下。

(1) 在 main 或其他方法中编写几行代码，并进行测试。
(2) 如果能够正确运行，就将它们封装到一个方法中，并再次测试。
(3) 如果这个方法没问题，就将其中的字面量替换为变量和形参。

其中的第二步被称为*封装*（encapsulation），第 3 步被称为*泛化*（generalization）。

为演示这种流程，我们将开发几个显示乘法表的方法。下面的循环显示 2 的幂，这些幂值都显示在一行中：

```
int i = 1;
while (i <= 6) {
    System.out.printf("%4d", 2 * i);
```

```
    i = i + 1;
}
System.out.println();
```

第 1 行初始化变量 `i`，这个变量将充当循环变量（loop variable）：每次执行循环时，都将变量 `i` 的值加 1；循环在 `i` 为 7 后终止。

每次循环都显示 `2 * i` 的值，并在它前面添加空格，使结果为 4 字符宽。由于我们使用的是 `System.out.printf`，因此所有输出都显示在一行中。

循环结束后，我们调用 `println` 打印一个换行符，从而换到下一行。别忘了，输出在有些环境中要遇到换行符后才显示。

因此，上述代码的输出如下：

```
2   4   6   8  10  12
```

下一步是将这些代码“封装”到一个新方法中。这个方法类似于下面这样：

```
public static void printRow() {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", 2 * i);
        i = i + 1;
    }
    System.out.println();
}
```

接下来，我们用一个形参 `n` 替换常量值 `2`。这一步被称为泛化，因为它让这个方法更通用（不那么具体）。

```
public static void printRow(int n) {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}
```

如果用实参 2 调用这个方法，得到的输出将与前面相同。使用实参 3 调用这个方法时，输出如下：

```
3   6   9  12  15  18
```

下面是用实参 4 调用这个方法得到的输出：

```
4   8  12  16  20  24
```

至此你可能猜到了我们将如何显示乘法表：反复调用方法 printRow，但每次指定不同的实参。实际上，我们将用另一个循环来遍历所有行。

```
int i = 1;
while (i <= 6) {
    printRow(i);
    i = i + 1;
}
```

输出类似于以下这样：

```
1   2   3   4   5   6
2   4   6   8  10  12
3   6   9  12  15  18
4   8  12  16  20  24
5  10  15  20  25  30
6  12  18  24  30  36
```

printRow 中的格式说明符 %4d 确保了输出都是垂直对齐的，不管结果是个位数还是十位数。

最后，我们将第二个循环封装到一个方法中：

```
public static void printTable() {
    int i = 1;
    while (i <= 6) {
        printRow(i);
        i = i + 1;
    }
}
```

编程时面临的挑战之一是如何将程序划分成多个方法，对初学者来说尤其如此。封装和泛化流程能够让你一边编程一边设计。

7.4　再谈泛化

前面的 printTable 版本总是显示 6 行，我们可对其进行泛化，用形参替换字面量 6：

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
        printRow(i);
        i = i + 1;
    }
}
```

下面是用实参 7 调用这个方法得到的输出：

```
1   2   3   4   5   6
2   4   6   8  10  12
3   6   9  12  15  18
4   8  12  16  20  24
5  10  15  20  25  30
6  12  18  24  30  36
7  14  21  28  35  42
```

这个版本更好，但还存在一个问题：每次显示的列数相同。要想进一步泛化，可给 printRow 再添加一个形参：

```
public static void printRow(int n, int cols) {
    int i = 1;
    while (i <= cols) {
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}
```

现在 printRow 接受两个形参——n 和 cols，n 指定要计算哪个值的整数倍，而 cols 指定列数。因为我们给 printRow 添加了一个形参，所以还需要修改 printTable 中调用 printRow 的代码行：

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
        printRow(i, rows);
        i = i + 1;
    }
}
```

这行代码执行时会将 rows（这里为 7）作为实参传递给 printRow。在 printRow 中，这个值被赋给 cols，这导致列数等于行数，因此得到的是一个 7×7 的方形表格：

```
1   2   3   4   5   6   7
2   4   6   8  10  12  14
3   6   9  12  15  18  21
4   8  12  16  20  24  28
5  10  15  20  25  30  35
6  12  18  24  30  36  42
7  14  21  28  35  42  49
```

合理地泛化方法常常会得到计划外的功能。例如，你可能注意到了，前面的乘法表是对称的，这是因为 $ab=ba$，所以这个表格中的每项都出现了两次。为节省油墨，可只打印这个表格的一半，为此只需要修改 printTable 中的一行代码：

```
printRow(i, i);
```

这使得每行的长度与其行号相同，结果是一个三角形乘法表：

```
1
2   4
3   6   9
4   8  12  16
5  10  15  20  25
6  12  18  24  30  36
7  14  21  28  35  42  49
```

泛化可让代码更通用、更易于重用甚至更容易编写。

7.5 for语句

前面编写的循环有几个共同之处：都先初始化一个变量，都有一个依赖于这个变量的条件，且都在循环体内修改这个变量。这种循环很常见，为了以更简洁的方式表示，Java 提供了另一条语句——for 循环。

例如，我们可以将 printTable 重写为下面这样：

```
public static void printTable(int rows) {
    for (int i = 1; i <= rows; i = i + 1) {
        printRow(i, rows);
    }
}
```

for 循环在括号内包含 3 个由分号分隔的部分：初始化部分、条件部分和更新部分。

(1) *初始化部分*只在循环开始时运行一次。
(2) 每次循环前都检查*条件部分*，如果它为 false，循环将终止；否则就再次执行循环。
(3) 每次迭代结束时，都运行*更新部分*再返回到第 2 步。

通常来说，for 循环更容易理解，因为它将所有与循环相关的语句都放在了循环开头。

while 循环和 for 循环的一个不同之处在于：若在初始化部分声明了变量，那么该变量只在 for 循环中可用。例如，下面是一个使用 for 循环的 printRow 版本：

```
public static void printRow(int n, int cols) {
    for (int i = 1; i <= cols; i = i + 1) {
        System.out.printf("%4d", n * i);
    }
    System.out.println(i); // 编译错误
}
```

最后一行试图显示 i（这样做只是为了演示），但行不通。要想在循环外使用循环变量，就必须在循环外声明它，如下所示：

```
public static void printRow(int n, int cols) {
    int i;
    for (i = 1; i <= cols; i = i + 1) {
        System.out.printf("%4d", n * i);
    }
```

```
        System.out.println(i);
    }
```

在 for 循环中，很少用 i = i + 1 这样的赋值语句，因为 Java 提供了表示加 1 和减 1 的更简洁方式。具体地说，++ 是*递增*（increment）运算符，与 i = i + 1 等效；而 -- 是*递减*（decrement）运算符，与 i = i - 1 等效。

如果给变量加上或减去的值不是 1，而是其他值，可用运算符 += 或 -=。例如，i += 2 表示将变量 i 加 2。

7.6　do-while循环

while 语句和 for 语句都是*先测试循环*（pretest loop），即在每次循环前都要测试条件。

Java 还提供了一种*后测试循环*（posttest loop）——do-while 语句。在至少需要运行循环一次时，这种循环很有用。

例如，在 5.7 节中用了 return 语句来避免读取无效的用户输入。我们也可以用 do-while 循环不断读取输入，直到用户输入有效为止：

```
Scanner in = new Scanner(System.in);
boolean okay;
do {
    System.out.print("Enter a number: ");
    if (in.hasNextDouble()) {
        okay = true;
    } else {
        okay = false;
        String word = in.next();
        System.err.println(word + " is not a number");
    }
} while (!okay);
double x = in.nextDouble();
```

这些代码看似很复杂，但其实只包含三个步骤。

(1) 显示提示。

(2) 检查输入。如果无效就显示错误消息并重新开始。

(3) 读取输入。

上述代码使用了标志，变量 okay 来指出是否要再次执行循环体。如果 hasNextDoublec() 返回 false，那么就调用 next() 来提取无效输入，再通过 System.err 显示一条错误消息，hasNextDouble() 返回 true 后终止循环。

7.7　break和continue

在有些情况下，先测试循环和后测试循环都无法完全满足需求。前一个示例就需要在循环

中间进行测试，因此我们结合使用了一个标志变量和一个嵌套的 if-else 语句。

对于这个问题，更简单的解决方案是使用 break 语句。程序执行到 break 语句时退出当前循环。

```
Scanner in = new Scanner(System.in);
while (true) {
    System.out.print("Enter a number: ");
    if (in.hasNextDouble()) {
        break;
    }
    String word = in.next();
    System.err.println(word + " is not a number");
}
double x = in.nextDouble();
```

在 while 循环中将 true 用作循环条件是一种惯用法，通常意味着不断循环，但在这个示例中意味着不断循环，直到遇到 break 语句。

除退出循环的 break 语句外，Java 还提供了直接进入下一次迭代的 continue 语句。例如，下面的代码不断从键盘读取整数，并计算这些整数的总和；其中的 continue 语句让程序忽略所有的负数。

```
Scanner in = new Scanner(System.in);
int x = -1;
int sum = 0;
while (x != 0) {
    x = in.nextInt();
    if (x <= 0) {
        continue;
    }
    System.out.println("Adding " + x);
    sum += x;
}
```

虽然 break 语句和 continue 语句让你能够更好地控制循环的执行流程，但也可能导致代码难以理解和调试，因此务必慎用。

7.8 术语表

- 迭代

  反复地执行一系列语句。

- 循环

  反复地执行一系列语句的语句。

- 循环体

  循环中的语句。

- *无限循环*

  条件始终为 `true` 的循环。

- *程序开发*

  程序编写流程，至此，我们介绍了两种程序开发流程——渐进开发以及封装和泛化。

- *封装*

  将一系列语句放在方法中。

- *泛化*

  将具体的信息（如常量值）替换为通用信息（如变量或形参）。

- *循环变量*

  为控制循环而被初始化、测试和修改的变量。

- *递增*

  增大变量的值。

- *递减*

  减小变量的值。

- *先测试循环*

  在每次迭代前检查条件的循环。

- *后测试循环*

  在每次迭代后检查条件的循环。

7.9　练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch07 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

如果你还没有阅读 A.5 节，那么现在正是阅读的好时机。该节介绍了 Checkstyle——一款分析源代码众多方面的工具。

练习7-1

请看下面的方法，并思考问题：

```
public static void main(String[] args) {
    loop(10);
}
```

```
public static void loop(int n) {
    int i = n;
    while (i > 1) {
        System.out.println(i);
        if (i % 2 == 0) {
            i = i / 2;
        } else {
            i = i + 1;
        }
    }
}
```

(1) 绘制一个表格，用来显示变量 `i` 和 `n` 在循环期间的值。在这个表格中，每次迭代要占据一行，每列对应一个变量。
(2) 这个程序的输出是什么？
(3) 只要 `n` 的值为正，这个循环就终将结束。你能证明这一点吗？

练习7-2

给定数字 a，要求你计算它的平方根。一种解决方案是先粗略地猜测结果 x_0，再用下面的公式提高结果的精度：

$$x_1 = (x_0 + a/x_0)/2$$

例如，要计算 9 的平方根，可从 x_0=6 开始，再用前面的公式计算 x_1=（6+9/6）/2 = 3.75，这更接近于准确的结果。我们可重复这个过程，根据 x_1 计算 x_2，再依次类推。就此例而言，x_2=3.075，x_3=3.00091。这种计算的速度非常快，很快就能找到正确的答案。

请编写一个名为 `squareRoot` 的方法，让它接受一个 `double` 值，并用前面的技巧计算其平方根的近似值。可不能用 `Math.sqrt` 哟！

为计算初始结果，可使用公式 $a/2$。这个方法应该不断迭代，直到两个相邻近似结果的差小于 0.0001。可用 `Math.abs` 计算差的绝对值。

练习7-3

在练习 6-9 中，你编写了一个以迭代方式计算幂的方法，它接受 `double` 值 `x` 和整数值 `n`，并返回 x^n。现在请编写一个迭代方法来执行这种计算。

练习7-4

6.7 节中介绍了一个计算阶乘的递归方法，请编写方法 `factorial` 的迭代版本。

练习7-5

要想计算 e^x 的值，一种办法是使用如下的无穷级数展开：

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \cdots$$

在上述级数中，第 i 项为“$x^i/i!$”。

(1) 编写一个名为 `myexp` 的方法，它接受形参 `x` 和 `n`，并将上述级数的前 `n` 项相加来计算 e^x 的近似值。为计算阶乘，可使用 6.7 节中的方法 `factorial`，也可使用前一个练习中编写的迭代版本。
(2) 在上述级数中，每一项的分子都是前一项的分子乘以 `x`，而每一项的分母都是前一项的分母乘以 `i`。利用这一点可避免使用方法 `Math.pow` 和 `factorial`，从而提高这个方法的效率。请按这种方式修改你在第 1 步编写的方法，并确定得到的结果相同。
(3) 编写一个名为 `check` 的方法，它接受形参 `x`，并显示 `x`、`myexp(x)` 和 `Math.exp(x)` 的值，其输出类似于下面这样：

```
1.0    2.708333333333333    2.718281828459045
```

要想用制表符分隔各列的值，可使用转义序列 `"\t"`。

(4) 修改级数包含的项数（`check` 向 `myexp` 传递的第二个实参），并查看这种修改对结果准确度的影响。在 `x` 为 1 的情况下调整值，直到估算值与正确的结果相同为止。
(5) 在方法 `main` 中编写一个循环，依次用实参值 0.1、1.0、10.0 和 100.0 调用 `check`。随着 `x` 值不断变化，结果的准确度将如何变化？比较估算值和实际值有多少位相同。
(6) 在方法 `main` 中编写一个循环，依次用实参值−0.1、−1.0、−10.0 和−100.0 调用 `check`，看看准确度将如何变化。

练习7-6

要想计算 $\exp(-x^2)$ 的值，一种办法是用如下的无穷级数展开：

$$\exp(-x^2) = 1 - x^2 + x^4/2 - x^6 6 + \ldots$$

在上述级数中，第 i 项为“$(-1)^i x^{2i}/i!$”。请编写一个名为 `gauss` 的方法，它接受形参 `x` 和 `n`，并返回上述级数中前 `n` 项的和。编写这个方法时，可不能使用方法 `factorial` 和 `pow` 哟！

第 8 章

数组

到目前为止，我们使用的变量只能存储单个值，如数字或字符串。你将在本章中学习如何用单个变量存储多个同类型的值。这种语言功能让你能够编写程序以便操作大量的数据。

8.1 创建数组

数组（array）包含一系列的值，这些值被称为*元素*（element）。你可以创建 `int` 数组、`double` 数组或其他任何类型的数组，但在同一个数组中，所有值的类型必须相同。

要想创建数组，必须先声明*数组类型*的变量，再创建数组本身。数组类型与其他 Java 类型看起来很像，但后面跟着方括号（[ ]）。例如，下面的代码行将 `counts` 和 `values` 分别声明为 `int` 数组和 `double` 数组：

```
int[] counts;
double[] values;
```

要创建数组本身，必须使用 3.2 节中首次见到的运算符 `new`：

```
counts = new int[4];
values = new double[size];
```

第 1 条赋值语句让 `count` 指向一个包含 4 个整数的数组；第 2 条语句让 `values` 指向一个 `double` 数组，而 `values` 中包含的元素数取决于 `size` 的值。

当然，也可在同一个代码行内声明变量并创建数组：

```
int[] counts = new int[4];
double[] values = new double[size];
```

可用任何整数表达式指定数组的长度，只要值不为负即可。例如，如果你试图创建一个包含-4 个元素的数组，将引发 NegativeArraySizeException 异常。数组可不包含任何元素，这种数组有其特殊用途，我们将在后面介绍。

8.2 访问元素

创建 int 数组时，其元素默认初始化为 0，图 8-1 是前面创建的数组 counts 的状态图。

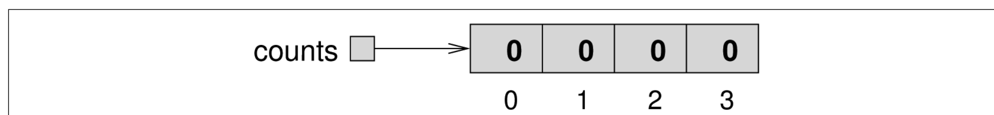


图 8-1：一个 int 数组的状态图

箭头表明 counts 的值是一个指向数组的引用（reference）。你应将数组和指向它的变量视为两码事，稍后你将看到，既可以给另一个变量赋值，使其与 counts 指向同一个数组，也可以修改 counts 的值，使其指向另一个数组。

方框内的数字表示的是数组的元素，方框外的数字是索引（index），用于标识数组中的各个位置。注意，第一个元素的索引为 0，而不是你预期的 1。

运算符 [] 用于选择数组中的元素：

```
System.out.println("The zeroth element is " + counts[0]);
```

可在表达式的任何地方使用运算符 []：

```
counts[0] = 7;
counts[1] = counts[0] * 2;
counts[2]++;
counts[3] -= 60;
```

图 8-2 显示了执行这些语句的结果。

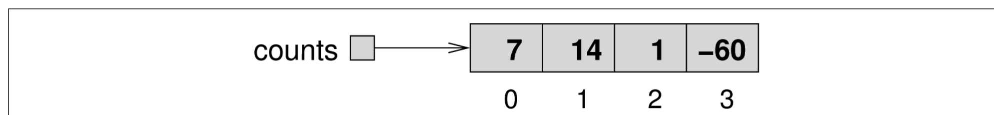


图 8-2：执行多条赋值语句后的状态图

可将任何表达式用作索引，只要其类型为 int。最常见的做法之一是将循环变量用作数组的索引，如下所示：

```
int i = 0;
while (i < 4) {
    System.out.println(counts[i]);
    i++;
}
```

while 循环从 0 数到 4；i 为 4 时不满足条件，循环结束。因此，仅在 i 为 0、1、2 和 3 时，循环体才被执行。

我们在每次的循环中都将 i 用作数组的索引，以显示第 i 个元素。这种数组处理方式通常是用 for 循环来实现的：

```
for (int i = 0; i < 4; i++) {
    System.out.println(counts[i]);
}
```

对数组 counts 来说，只有索引 0、1、2 和 3 是合法的。如果索引为负或大于 3，将引发 ArrayIndexOutOfBoundsException 异常。

8.3 显示数组

可用 println 显示数组，但结果很可能不是你所希望的。例如，下面的代码片段声明了一个数组变量，让它指向一个包含 4 个元素的数组，并试图用 println 显示这个数组的内容：

```
int[] a = {1, 2, 3, 4};
System.out.println(a);
```

遗憾的是，输出类似于以下这样：

```
[I@bf3f7e0
```

方括号表明值是一个数组，I 表示整数，余下的内容是这个数组的地址。要想显示数组的元素，需要分别显示它们：

```
public static void printArray(int[] a) {
    System.out.print("{" + a[0]);
    for (int i = 1; i < a.length; i++) {
        System.out.print(", " + a[i]);
    }
    System.out.println("}");
}
```

给定前面的数组，该方法的输出如下：

```
{1, 2, 3, 4}
```

Java 库包含实用工具类 java.util.Arrays，这个类提供了处理数组的方法。其中一个方法是 toString，用于返回数组的字符串表示。你可以像下面这样调用：

```
System.out.println(Arrays.toString(a));
```

输出如下：

```
[1, 2, 3, 4]
```

与往常一样，必须先导入才能使用 `java.util.Arrays`。注意，字符串格式与前面的输出格式稍有不同，它使用方括号而不是大括号。但以这种格式输出时无需编写方法 `printArray`。

8.4 复制数组

8.2 节中说过，数组变量包含指向数组的*引用*。给数组变量赋值时，只复制指向数组的引用，而不会复制数组本身！请看下面的示例：

```
double[] a = new double[3];
double[] b = a;
```

这些语句创建了一个包含 3 个元素的 `double` 数组，并让两个变量指向它，如图 8-3 所示。

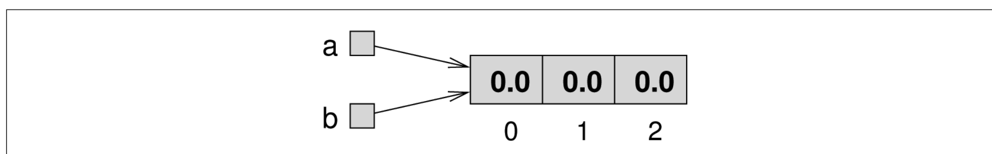


图 8-3：显示两个变量指向同一个数组的状态图

通过其中任何一个变量修改数组都将影响另一个变量。例如，如果我们设置 `a[0] = 17.0`，再显示 `b[0]`，结果将为 `17.0`。因为 `a` 和 `b` 是表示同一样东西的不同名称，所以它们有时被称为*别名*（alias）。

如果要复制数组本身而不是指向它的引用，那么就必须创建一个新的数组，并将旧数组中的每个元素都复制到新数组中，如下所示：

```
double[] b = new double[3];
for (int i = 0; i < 3; i++) {
    b[i] = a[i];
}
```

另一种选择是使用 `java.util.Arrays`，它提供了复制数组的方法 `copyOf`，可像下面这样调用这个方法：

```
double[] b = Arrays.copyOf(a, 3);
```

其中的第二个参数用来指定要复制的元素个数，因此你也可以只复制数组的一部分。

8.5 数组的长度

前面的示例仅在数组包含 3 个元素时才管用，因此，最好对这些代码进行泛化，使其适用于任何长度的数组。为此，可将魔幻数字 3 替换为 a.length：

```
double[] b = new double[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

所有数组都有一个内置常量 length，其中存储了数组包含的元素数。表达式 a.length 看起来像方法调用，但没有括号，也没有实参。

这个循环最后一次执行时，i 的值为 a.length-1，这是最后一个元素的索引。i 的值为 a.length 时不满足条件，因此不会执行循环体，这是件好事，因为试图访问 a[a.length] 将引发异常。

还可将 a.length 用作 Arrays.copyOf 的第二个实参：

```
double[] b = Arrays.copyOf(a, a.length);
```

8.6 数组遍历

很多计算可通过循环访问数组的每个元素并对其执行特定的操作来实现。例如，下面的循环计算了一个 double 数组的各个元素的平方：

```
for (int i = 0; i < a.length; i++) {
    a[i] = Math.pow(a[i], 2.0);
}
```

循环访问数组的每个元素被称为*遍历*（traversal）。另一种常见的模式是*查找*（search），这需要遍历数组，在其中查找特定的元素。

例如，下面的方法接受一个 int 数组和一个 int 值，并返回 int 值在数组中所处位置的索引：

```
public static int search(double[] a, double target) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == target) {
            return i;
        }
    }
    return -1;
}
```

如果在数组中找到目标值，那么就立即返回其索引；如果循环结束时也没有找到目标值，

那么就返回 `-1`——用于表示查找失败的特殊值。

另一种常见的遍历是*归并*（reduce）操作，用于将数组归并为单个值。归并操作包括计算各个元素的和或积、找出最大值或最小值。下面的方法接受一个 `double` 数组，并返回其中所有元素的和：

```
public static int sum(double[] a) {
    double total = 0.0;
    for (int i = 0; i < a.length; i++) {
        total += a[i];
    }
    return total;
}
```

循环前将 `total` 初始化为零。在每次循环时都将 `total` 加上一个数组元素的值；循环结束后，`total` 为数组中所有元素的和。以这种方式使用的变量有时被称为*累加器*（accumulator）。

8.7 随机数

大多数计算机程序每次运行时所做的事情相同，这样的程序是*确定的*（deterministic）。确定性是件好事，因为我们希望同样的计算得到同样的结果。但对有些应用程序来说，我们希望它们的行为是不可预测的，游戏就是一个显而易见的例子，当然还有很多其他的例子。

实际上，很难让程序的行为是*不确定的*（nondeterministic），因为要让计算机生成真正的随机数很难。但存在一些算法，可用于生成被称为*伪随机数*（pseudorandom number）的不可预测序列。对大多数应用程序来说，伪随机数的随机程度已经足够高了。

如果你完成了练习 3-4，就会知道 `java.util.Random` 生成的就是伪随机数。这个类的方法 `nextInt` 接受 `int` 实参 `n`，并返回一个位于 `0~n-1`（闭区间）的随机数。

如果你生成大量的随机数，每个值出现的次数将大致相同。要验证 `nextInt` 的这种行为，可以用它生成大量的随机数，然后将这些随机数存储在一个数组中，并计算每个值出现的次数。

下面的方法创建了一个 `int` 数组，并用 0~99 的随机数来填充它。数组的长度由传递的实参指定，且返回值是一个引用，指向新创建的数组。

```
public static int[] randomArray(int size) {
    Random random = new Random();
    int[] a = new int[size];
    for (int i = 0; i < a.length; i++) {
        a[i] = random.nextInt(100);
    }
```

```
        return a;
    }
```

下面的代码片段创建了一个数组，并用 8.3 节的方法 `printArray` 显示这个数组：

```
    int numValues = 8;
    int[] array = randomArray(numValues);
    printArray(array);
```

输出类似于以下这样：

```
    {15, 62, 46, 74, 67, 52, 51, 10}
```

运行这些代码生成的随机数可能与这里显示的不同。

8.8　遍历和计数

如果前面的值表示的是考试成绩，并且这些成绩非常糟糕，老师可能会在课程上以直方图（histogram）的方式呈现它们。直方图在统计学中是一组计数器，记录了每个值出现的次数。

就考试成绩而言，我们可能需要 10 个计数器，用来指出考试成绩为 90~100 分、80~90 分等的学生分别有多少。为此，我们可以遍历数组，并计算值在给定范围内的元素数。

下面的方法接受一个数组和两个整数（`low` 和 `high`），并返回值在范围 `low~high` 的元素数。

```
    public static int inRange(int[] a, int low, int high) {
        int count = 0;
        for (int i = 0; i < a.length; i++) {
            if (a[i] >= low && a[i] < high) {
                count++;
            }
        }
        return count;
    }
```

你应该很熟悉这种模式：这也是一种归并操作。注意，范围包含 `low`（>=），但不包含 `high`（<）。这种细节可避免我们将一个分数算两次。

现在可以计算每个范围内的考试成绩数了：

```
    int[] scores = randomArray(30);
    int a = inRange(scores, 90, 100);
    int b = inRange(scores, 80, 90);
    int c = inRange(scores, 70, 80);
    int d = inRange(scores, 60, 70);
    int f = inRange(scores, 0, 60);
```

8.9 生成直方图

前面的代码有些重复，但只要范围数不多，这是完全可以接受的。然而，如果我们要计算每个分数出现的次数，就必须编写 100 行代码：

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
...
int count99 = inRange(scores, 99, 100);
```

我们需要存储 100 个计数器，最好还能用索引来访问计数器。换言之，我们需要一个数组！

下面的代码片段创建了一个包含 100 个计数器的数组，每种可能的分数一个。它遍历分数，并用 `inRange` 计算每种分数出现的次数，并将结果存储在数组中：

```
int[] counts = new int[100];
for (int i = 0; i < counts.length; i++) {
    counts[i] = inRange(scores, i, i + 1);
}
```

注意，我们在三个地方使用了循环变量 `i`：一处用作索引访问数组 `counts`，两处用于设置 `inRange` 的两个实参。这些代码可行，但在效率方面还有改进空间。这个循环每次调用 `inRange` 时都遍历整个数组。

更好的做法是只遍历数组一次；计算每个成绩所在的范围，并将相应的计数器加 1。下面的代码在只遍历数组一次的情况下生成直方图：

```
int[] counts = new int[100];
for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

这个循环每次执行时，都选择数组 `scores` 中的一个元素，并将其用作索引来将数组 `counts` 中相应的元素加 1。因为这些代码只遍历数组 `scores` 一次，所以其效率高得多。

8.10 改进的for循环

考虑到遍历数组的操作极其常见，Java 提供了一种让代码更紧凑的语法。例如，请看下面的 `for` 循环，它将数组的每个元素都单独显示在一行中：

```
for (int i = 0; i < values.length; i++) {
    System.out.println(values[i]);
}
```

对于这个循环，我们可将其重写成下面这样：

```
for (int value : values) {
    System.out.println(value);
}
```

这条语句被称为*改进的 for 循环*（enhanced for loop），你可将其解读为“对于 values 中的每个值 value”。根据约定，数组变量应使用复数名词，元素变量应使用单数名词。

通过使用改进的 for 循环并删除临时变量，我们可用更简洁的方式编写前一节生成直方图的代码：

```
int[] counts = new int[100];
for (int score : scores) {
    counts[score]++;
}
```

改进的 for 循环通常可提高代码的可读性，对计算累积值的代码来说尤其如此。但如果需要引用索引（如执行查找操作时），这种 for 循环就并没有太大的用处了。

8.11 术语表

- *数组*

  一系列的值，所有值的类型都相同，每个值都由一个索引标识。

- *元素*

  数组中的一个值；可用运算符 [] 选择元素。

- *索引*

  标识数组元素的 int 变量或 int 值。

- *引用*

  标识另一个值（如数组）的值；在状态图中用箭头表示。

- *别名*

  与另一个变量指向同一个对象的变量。

- *遍历*

  通过循环访问数组（或其他集合）中的每个元素。

- *查找*

  一种在数组中查找特定元素的遍历模式。

- *归并*

  一种将数组的所有元素合并为单个值的遍历模式。

- 累加器

  在遍历期存储累积结果的变量。

- 确定的

  每次运行时结果都相同的程序。

- 不确定的

  每次运行时结果都不同的程序，即便每次运行时输入相同亦如此。

- 伪随机数

  一系列看似随机，但实际是用确定性计算得到的数字。

- 直方图

  一个 `int` 数组，其中的每个元素指出了有多少个值位于特定范围内。

- 改进的for循环

  另一种遍历数组元素（值）的语法。

8.12 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch08 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习8-1

这个练习旨在用本章的一些示例来练习封装。

(1) 以 8.6 节中的代码为基础，编写一个名为 `powArray` 的方法，让它接受一个 `double` 数组 `a`，并返回一个新数组，其中包含数组 `a` 中的所有元素的平方。泛化这个方法使其接受另一个参数，该参数要指定计算元素的几次方。

(2) 以 8.10 节的代码为基础，编写一个名为 `histogram` 的方法，让它接受一个 `int` 数组，其中存储了 0~100（不包括 100）的成绩，并返回一个包含 100 个计数器的直方图。泛化这个方法使其接受另一个指定计数器数量的参数。

练习8-2

这个练习旨在让你阅读代码，并识别本章介绍过的遍历模式。下面的方法难以阅读，因为它们给变量指定的不是有意义的名称，而是水果名。

```
public static int banana(int[] a) {
    int kiwi = 1;
    int i = 0;
    while (i < a.length) {
        kiwi = kiwi * a[i];
        i++;
    }
```

```
        return kiwi;
    }

    public static int grapefruit(int[] a, int grape) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] == grape) {
                return i;
            }
        }
        return -1;
    }

    public static int pineapple(int[] a, int apple) {
        int pear = 0;
        for (int pine: a) {
            if (pine == apple) {
                pear++;
            }
        }
        return pear;
    }
```

用一句话描述每个方法的功能，无需涉及其相关的工作原理细节。指出每个变量扮演的角色。

练习8-3

下述程序的输出是什么？请绘制一个栈图，以显示该程序在 `mus` 返回前的状态。用几个词描述 `mus` 的功能。

```
    public static int[] make(int n) {
        int[] a = new int[n];
        for (int i = 0; i < n; i++) {
            a[i] = i + 1;
        }
        return a;
    }

    public static void dub(int[] jub) {
        for (int i = 0; i < jub.length; i++) {
            jub[i] *= 2;
        }
    }

    public static int mus(int[] zoo) {
        int fus = 0;
        for (int i = 0; i < zoo.length; i++) {
            fus += zoo[i];
        }
        return fus;
    }

    public static void main(String[] args) {
```

```
        int[] bob = make(5);
        dub(bob);
        System.out.println(mus(bob));
    }
```

练习8-4

编写一个名为 indexOfMax 的方法，让它接受一个 int 数组，并返回其中最大元素的索引。你能用改进的 for 循环来编写这个方法吗？为什么？

练习8-5

埃拉托斯特尼筛法是一种古老而简单的算法，用于找出小于指定值的所有素数。这种算法的相关详细信息请参阅 https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes。

请编写一个名为 sieve 的方法，让它接受一个 int 参数 n，并返回一个 boolean 数组，指出 0~n–1 的各个数是否为素数。

练习8-6

编写一个名为 areFactors 的方法，让它接受一个 int 参数 n 和一个 int 数组，并在这个数组的所有元素都是 n 的因数（即 n 可被所有元素整除）时返回 true。

练习8-7

编写一个名为 arePrimeFactors 的方法，让它接受一个 int 参数 n 和一个 int 数组，并在这个数组的所有元素都为素数且它们的乘积为 n 时返回 true。

练习8-8

本章介绍的很多数组遍历模式也可以用递归方式来实现。这种做法虽然不常见，却是一个很有用的练习。

(1) 编写一个名为 maxInRange 的方法，让它接受一个 int 数组和两个索引（lowIndex 和 highIndex），并在这两个索引指定的范围内（闭区间）找出最大的元素。

这个方法必须是递归的。如果范围为 1，即 lowIndex==highIndex，那么就意味着这个范围只包含一个元素，因此它就是最大的。所以这是基线条件。

如果这个范围包含多个元素，我们可将这个范围分成两部分，并在两部分中分别找出最大的元素，再在这两个最大的元素中找出更大的那个。

(2) maxInRange 这样的方法可能难以使用。要找出数组中的最大元素，我们必须将范围指定为整个数组。

```
double max = maxInRange(a, 0, a.length - 1);
```

编写一个名为 max 的方法，让它接受一个数组，然后用 maxInRange 找出并返回其中的最大元素。

第 9 章

字符串

在 Java 和其他面向对象的语言中，对象（object）是提供一系列方法的数据集合。例如，3.2 节介绍的 Scanner 就是提供输入分析方法的对象，而 System.out 和 System.in 也都是对象。

字符串也是对象。它们包含字符，并提供了操作字符数据的方法。本章将探索其中几个方法。

在 Java 中，并非什么都是对象。例如，int、double 和 boolean 都是所谓的基本类型（primitive type）。本章将阐述对象和基本类型之间的一些差别。

9.1 字符

字符串提供了提取字符的方法 charAt，这个方法会返回一个 char，这是一种存储单个字符（而不是字符串）的基本类型。

```
String fruit = "banana";
char letter = fruit.charAt(0);
```

实参 0 表示要提取位置 0 处的字符。与数组索引一样，字符串索引也从 0 开始，因此，赋给变量 letter 的字符是字母 b。

字符的工作原理与前面介绍过的其他基本类型相似，可用关系运算符来比较它们：

```
if (letter == 'a') {
    System.out.println('?');
}
```

字符字面量是用单引号括起的，如 'a'。不同于用双引号括起的字符串字面量，字符字面量只能包含一个字符。转义序列（如 '\t'）是合法的字符字面量，因为它们表示的是单个字符。

递增和递减运算符也可用于字符，因此下面的循环显示字母表中的所有字母：

```
System.out.print("Roman alphabet: ");
for (char c = 'A'; c <= 'Z'; c++) {
    System.out.print(c);
}
System.out.println();
```

Java 用 Unicode 表示字符，因此字符串可存储西里尔文字和希腊文字，还可存储非字母文字（如中文），有关这方面的更多详细信息请参阅 http://unicode.org/。

在 Unicode 中，每个字符由一个字符编码表示，我们可将字符编码视为整数。大写的希腊字母的字符编码为 913~937，因此我们可以像下面这样显示希腊字母表：

```
System.out.print("Greek alphabet: ");
for (int i = 913; i <= 937; i++) {
    System.out.print((char) i);
}
System.out.println();
```

这个示例使用了类型转换将指定范围内的每个整数都转换为相应的字符。

9.2 字符串是不可修改的

字符串提供了方法 toUpperCase 和 toLowerCase，它们可分别转换为大写和小写。这些方法常常令人迷惑，因为它们好像修改了字符串，但实际上，这些方法以及其他字符串操作方法都不能修改字符串，因为字符串是*不可修改的*（immutable）。

对字符串调用 toUpperCase 将生成并返回一个新的字符串对象。请看下面的示例：

```
String name = "Alan Turing";
String upperName = name.toUpperCase();
```

这些语句执行后，upperName 将指向字符串 "ALAN TURING"，但 name 依然指向字符串 "Alan Turing"。

另一个很有用的方法是 replace，它在字符串中查找并替换指定的子串。例如，下面的代码将 "Computer Science" 替换为 "CS"：

```
String text = "Computer Science is fun!";
text = text.replace("Computer Science", "CS");
```

这个示例演示了使用字符串方法的一种常见方式。它调用 text.replace，然后 text.replace 方法返回一个引用，该引用指向新字符串 "CS is fun!"。接下来，它将这个引用赋给变量 text，使其不再指向原来的字符串。

这个赋值操作很重要；如果不保存返回的值，调用 text.replace 将不会有任何影响。

9.3 字符串遍历

下面的循环遍历了字符串变量 fruit 中的字符，并以每个字符独占一行的方式显示：

```
for (int i = 0; i < fruit.length(); i++) {
    char letter = fruit.charAt(i);
    System.out.println(letter);
}
```

字符串提供了一个名为 length 的方法，该方法返回了字符串包含的字符数。因为这是一个方法，所以调用它时必须指定空的实参列表——()。

条件为 i<fruit.length()，这意味着当 i 与字符串长度相等时，这个条件为 false，循环将终止。

遗憾的是，改进的 for 循环不能用于遍历字符串。但你可以将任何字符串转换为字符数字，再用改进的 for 循环来迭代：

```
for (char letter : fruit.toCharArray()) {
    System.out.println(letter);
}
```

为获取字符串的最后一个字符，你可能试图像下面这样做：

```
int length = fruit.length();
char last = fruit.charAt(length);     // 不对!
```

这些代码能够通过编译并运行，但其中对方法 charAt 的调用将引发 StringIndexOutOfBoundsException 异常，这是因为 "banana" 没有索引为 6 的字符。由于索引从 0 开始，这 6 个字符的索引为 0~5。要获取最后一个字符，必须将 length 减 1。

```
int length = fruit.length();
char last = fruit.charAt(length - 1); // 正确
```

很多字符串遍历操作涉及读取一个字符串并创建另一个字符串。例如，要想反转字符串，可按从后到前的顺序将字符依次加入到另一个字符串中：

```
public static String reverse(String s) {
    String r = "";
    for (int i = s.length() - 1; i >= 0; i--) {
```

```
            r = r + s.charAt(i);
        }
        return r;
    }
```

r 的初始值为 ""，即空字符串（empty string）；其中的循环按从后到前的顺序遍历 s 中的所有字符。每次执行循环时都创建一个新的字符串，并将其赋给 r。循环结束时，r 包含 s 的所有字符，但排列顺序相反。因此，reverse("banana") 的结果为 "ananab"。

9.4　子串

方法 substring 返回一个新的字符串，其中包含已有字符串中从指定索引到末尾的字符。

- fruit.substring(0) 返回 "banana"
- fruit.substring(2) 返回 "nana"
- fruit.substring(6) 返回 ""

第一个示例返回整个字符串的副本；第二个示例返回除前两个字符之外的所有其他字符；最后一个示例表明，如果实参为字符串的长度，则 substring 将返回一个空字符串。

为理解方法 substring 的工作原理，绘制类似于图 9-1 所示的示意图大有裨益。

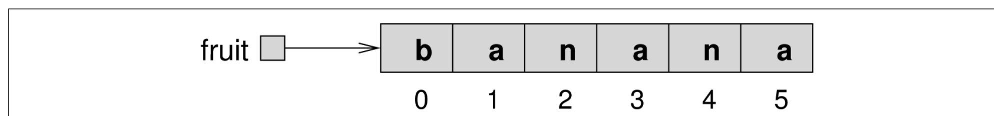


图 9-1：包含 6 个字符的字符串状态图

与大多数的字符串方法一样，substring 也被重载；也就是说，substring 还有接受不同参数的其他版本。用两个实参调用 substring 时，这两个实参将分别视为起始索引和终止索引：

- fruit.substring(0, 3) 返回 "ban"
- fruit.substring(2, 5) 返回 "nan"
- fruit.substring(6, 6) 返回 ""

注意，返回的字符串中不包含终止索引处的字符。substring 方法的这个版本简化了一些常见的操作。例如，要想从索引 i 处开始提取长度为 len 的子串，可编写代码 fruit.substring(i, i+len)。

9.5　方法indexOf

方法 indexOf 用于在字符串中查找字符。

```
String fruit = "banana";
int index = fruit.indexOf('a');
```

这个示例确定了字符 'a' 在字符串中的索引，但因为这个字符出现了三次，所以 indexOf 的结果是什么并不那么明显。文档指出，这个方法返回的是字符第一次出现处的索引。

要想确定后面位置出现的索引，可使用另一个版本的 indexOf，它接受第二个实参，指定从字符串的什么位置开始查找。

```
int index = fruit.indexOf('a', 2);
```

这些代码从索引 2（第一个 'n'）处开始查找下一个 'a'，这个 'a' 的索引为 3。如果要查找的字符刚好在起始索引处，结果将为起始索引，因此 fruit.indexOf('a', 5) 返回 5。

如果字符串中没有指定的字符，indexOf 将返回 -1。因为索引不可能为负，所以这个值表明没有找到指定的字符。

还可用 indexOf 查找子串，而不仅仅是单个字符。例如，表达式 fruit.indexOf("nan") 返回 2。

9.6 字符串比较

你可能想用运算符 == 和 != 来比较两个字符串：

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";
if (name1 == name2) {                 // 不对!
    System.out.println("The names are the same.");
}
```

这些代码能够通过编译并运行，且在大多数情况下能得到正确的答案。但这并不是正确的，有时得到的答案也不对。这是因为运算符 == 通过比较引用来判断两个变量指向的是否为同一个对象。如果你让它比较两个包含相同字符的字符串，结果将为 false。

要比较字符串，正确的做法是像下面这样使用方法 equals：

```
if (name1.equals(name2)) {
    System.out.println("The names are the same.");
}
```

这个示例对 name1 调用 equals，并将实参指定为 name2。如果两个字符串包含相同的字符，方法 equals 将返回 true，否则返回 false。

如果两个字符串不同，可用 compareTo 来确定按字母表顺序排列时哪个字符串在前：

```
int diff = name1.compareTo(name2);
if (diff == 0) {
    System.out.println("The names are the same.");
} else if (diff < 0) {
    System.out.println("name1 comes before name2.");
} else if (diff > 0) {
    System.out.println("name2 comes before name1.");
}
```

compareTo 的返回值为两个字符串中第一个不同的字符的差。如果两个字符串相等，则差为零；如果按字母表顺序排列时，第一个字符串（对其调用这个方法的字符串）在前，则差值为负，否则为正。

在前面的代码中，compareTo 返回 8，这是因为 "Ada" 的第二个字母比 "Alan" 的第二个字母靠前 8 个位置。

equals 和 compareTo 都区分大小写。因为大写字母排在小写字母前，所以 "Ada" 排在 "ada" 前。

9.7　设置字符串的格式

我们在 3.6 节中学习了如何用 printf 以特定的格式显示输出。在有些情况下，程序需要创建特定格式的字符串，但不马上显示它们，甚至根本不显示。例如，下面的方法返回了一个以 12 小时制表示时间的字符串：

```
public static String timeString(int hour, int minute) {
    String ampm;
    if (hour < 12) {
        ampm = "AM";
        if (hour == 0) {
            hour = 12;  // 午夜
        }
    } else {
        ampm = "PM";
        hour = hour - 12;
    }
    return String.format("%02d:%02d %s", hour, minute, ampm);
}
```

String.format 接受的参数与 System.out.printf 相同：一个格式说明符和一系列的值。主要的差别在于，System.out.printf 将结果显示到屏幕上，而 String.format 创建一个新的字符串，但什么都不显示。

在这个示例中，格式说明符 %02d 表示将整数显示为两位（不够两位就添加前导零），因此 timeString(19, 5) 返回字符串 "07:05 PM"。

9.8 包装类

基本类型（如 int、double 和 char）不提供方法。例如，你不能对 int 值调用 equals：

```
int i = 5;
System.out.println(i.equals(5));  // 编译错误
```

但 Java 库包含与每种基本类型对应的类，这些类被称为包装类（wrapper class）。与 char 对应的包装类为 Characte；与 int 对应的包装类为 Integer；其他包装类包括 Boolean、Long 和 Double。这些包装类都位于 java.lang 包中，因此无需导入就可使用。

每个包装类都定义了常量 MIN_VALUE 和 MAX_VALUE。例如，Integer.MIN_VALUE 的值为 −2147483648，而 Integer.MAX_VALUE 的值为 2147483647。因为包装类提供了这些常量，所以无需记住，也不用在程序中定义。

包装类提供了将字符串转换为其他类型的方法。例如，Integer.parseInt 将字符串转换为整数：

```
String str = "12345";
int num = Integer.parseInt(str);
```

这里的分析（parse）指的是读取并转换。

其他的包装类提供了类似的方法，如 Double.parseDouble 和 Boolean.parseBoolean。包装类还提供了方法 toString，它返回值的字符串表示：

```
int num = 12345;
String str = Integer.toString(num);
```

结果为字符串 "12345"。

9.9 命令行实参

本书前面一直对 main 方法的形参 args 置之不理，现在你已经熟悉了数组和字符串，我们终于可以说说 args 了。如果你不熟悉命令行界面，请阅读 A.3 节。

下面来编写一个程序以找出一系列数字的最大值。这里不从 System.in 读取数字，而是通过命令行实参来传递它们。这个程序的初始版本如下：

```
public class Max {
    public static void main(String[] args) {
        System.out.println(Arrays.toString(args));
    }
}
```

要运行这个程序，可在命令行中执行如下命令：

```
java Max
```

输出表明，`args` 是个空数组（empty array），即不包含任何元素：

```
[]
```

但如果在命令行中提供了额外的值，它们将作为实参传递给 `main`。例如，如果你像下面这样运行这个程序：

```
java Max 10 -3 55 0 14
```

输出如下：

```
[10, -3, 55, 0, 14]
```

别忘了，`args` 的元素为字符串。要想找出最大的数字，必须将这些实参转换为整数。

下面的代码片段结合使用了改进的 `for` 循环和包装类 `Integer` 来分析实参并找出最大的值：

```
int max = Integer.MIN_VALUE;
for (String arg : args) {
    int value = Integer.parseInt(arg);
    if (value > max) {
        max = value;
    }
}
System.out.println("The max is " + max);
```

`max` 的初始值为 `int` 类型可表示的最小值，因此任何其他 `int` 值都比它大。如果 `args` 为空数组，结果将为 `MIN_VALUE`。

9.10 术语表

- 对象
  一系列相关的数据以及一组操作这些数据的方法。

- 基本类型
  存储单个值且没有提供任何方法的数据类型。

- Unicode
  字符编码的一种标准，涵盖全球大部分语言中的字符。

- 不可修改的
  一旦创建就不能修改的对象。字符串就被设计成不可修改。

- 空字符串

  不包含任何字符且长度为零的字符串，用 "" 表示。

- 包装类

  `java.lang` 中的一些类，提供了处理基本类型的常量和方法。

- 分析

  读取字符串并对其进行解读或转换。

- 空数组

  不包含任何元素且长度为零的数组。

9.11 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch09 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习9-1

这个练习旨在探索 Java 类型，并补充本章前面未涉及的一些细节。

(1) 创建一个新程序，将其命名为 Test.java，并在 `main` 方法中编写一些用运算符 + 将不同数据类型“相加”的表达式。例如，如果将字符串和 `char`“相加”，结果将会如何？这会执行字符加法运算还是执行字符串拼接操作呢？结果是什么类型？你又是如何确定结果类型的？

(2) 复制并填写下面的表格。在任何两种类型的交叉位置指出对它们使用运算符 + 是否合法，如果合法，将执行什么样的操作（加法运算还是拼接操作）？结果是什么类型呢？

	boolean	char	int	double	string
boolean					
char					
int					
double					
string					

(3) 想想 Java 的设计者在填写这个表格时作出的一些选择。在这个表格中，有多少项因为别无选择而无法避免。又有多少项原本有同样充分理由的可能性，但 Java 的设计者只是随便选择了其中的一个。哪些项存在严重的问题？

(4) 通常情况下，语句 `x++` 与 `x = x + 1` 完全等价，但如果 `x` 为 `char`，情况就不是这样了。在这种情况下，`x++` 是合法的，但 `x = x + 1` 将导致错误。请尝试这样做，看看将出现什么样的错误消息，并试着找出错误的原因。

(5) 将 ""（空字符串）与其他类型的值相加（如 ""+5）时，结果将如何？

(6) 可将哪些类型的值赋给各种类型的变量？例如，可将 int 值赋给 double 变量，但反过来不行。

练习9-2

编写一个名为 letterHist 的方法，让它接受一个字符串参数，并返回一个表示该字符串各字母出现次数的直方图。在返回的直方图中，第 0 个元素为字母 a（不区分大小写）在这个字符串中出现的次数，第 25 个元素为字母 z 出现的次数。你的解决方案只能遍历该字符串一次。

练习9-3

这个练习旨在复习封装和泛化（参见 7.3 节）。下面的代码片段遍历了一个字符串，并检查了它包含的左括号数和右括号数是否相等：

```
String s = "((3 + 7) * 2)";
int count = 0;

for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (c == '(') {
        count++;
    } else if (c == ')') {
        count--;
    }
}

System.out.println(count);
```

(1) 请将这些代码封装到一个方法中，让这个方法接受一个字符串参数，并返回 count 的终值。

(2) 泛化这些代码使其适用于任何字符串，然后还能如何进一步泛化呢？

(3) 用多个字符串测试编写的方法，包括左右括号数相等和不等的字符串。

练习9-4

创建一个名为 Recurse.java 的程序，并在其中输入以下方法：

```
/**
 * 返回给定字符串中的第一个字符。
 */
public static char first(String s) {
    return s.charAt(0);
}

/**
 * 返回给定字符串中除第一个字符外的其他所有字符。
 */
public static String rest(String s) {
```

```
    return s.substring(1);
}

/**
 * 返回给定字符串中除第一个和最后一个字符外的其他所有字符。
 */
public static String middle(String s) {
    return s.substring(1, s.length() - 1);
}

/**
 * 返回给定字符串的长度。
 */
public static int length(String s) {
    return s.length();
}
```

(1) 在 main 中编写一些代码以测试上述的每个方法。确认它们能够正确地工作，并确保你明白它们的功能。

(2) 编写一个名为 printString 的方法，让它接受一个字符串参数，并显示这个字符串中的所有字符，且每个字符独占一行。编写这个方法时，只能用前面定义的方法，不能用其他字符串方法。另外，这个方法应为 void 方法。

(3) 编写一个名为 printBackward 的方法，其功能与 printString 相同，但按相反的顺序显示字符串中的字符，且每个字符也独占一行。同样，在编写这个方法时，你只能用前面定义的方法。

(4) 编写一个名为 reverseString 的方法，让它接受一个字符串参数，并返回一个新的字符串。这个新字符串包含的字符与参数字符串相同，但排列顺序相反。换言之，对于下述示例代码：

```
String backwards = reverseString("coffee");
System.out.println(backwards);
```

其输出应为：

```
eeffoc
```

(5) 回文指的是顺着读和倒着读一样的单词，如 otto 和 palindromeemordnilap。一种判断单词是否为回文的方式如下：

> 只包含一个字母的单词是回文；单词包含两个字母时，如果这两个字母相同，那么这个单词是回文；对于其他的单词，如果第一个字母和最后一个字母相同，且余下的部分为回文，则这个单词为回文。

请编写一个名为 isPalindrome 的递归方法，让它接受一个字符串，并返回一个 boolean 值来指出这个字符串是否为回文。

练习9-5

如果一个单词包含的字母是按字母表顺序排列的，那么这个单词就是 abecedarian 单词。例如，下面列出了所有按字母顺序排列的含 6 个字母的英语单词：

> abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy,cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

请编写一个名为 `isAbecedarian` 的方法，让它接受一个字符串，并返回一个 `boolean` 值，指出这个字符串表示的单词是否是按字母顺序排列的。编写的方法可以是迭代的，也可以是递归的。

练习9-6

如果一个单词包含的每个字母都刚好出现两次，那么它就是 doubloon 单词。下面是字典中的一些 doubloon 单词：

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

请编写一个名为 `isDoubloon` 的方法，让它接受一个字符串，并检查它是否为 doubloon 单词。为忽略大小写，可在检查前调用方法 `toLowerCase`。

练习9-7

如果两个单词包含的字母相同，且其中的每个字母出现的次数也相同，那么这两个单词就是重组词。例如，单词 stop 是 pots 的重组词，而 allen downey 是 well annoyed 的重组词。

请编写一个方法，让它接受两个字符串，并检查它们是否为重组词。

练习9-8

在拼字游戏 Scrabble 中，每个玩家都有一组字母卡片，玩家需要用这些卡片拼出单词。其中的计分系统非常复杂，但一般而言，拼出的单词越长，得分越高。

假设以一个字符串（如 `"quijibo"`）的方式指定了你手中有哪些字母卡片，并要求你判断能否用这些卡片拼出另一个字符串，如 `"jib"`。

请编写一个名为 `canSpell` 的方法，让它接受两个字符串，并判断用第一个字符串指定的字母卡片能否拼出第二个字符串指定的单词。可能会有多个包含相同字母的卡片，但每个卡片只能用一次。

第 10 章

对象

我们在前一章说过，对象是提供一组方法的数据集合。例如，`String` 是一个字符串集合，提供了 `charAt` 和 `substring` 等方法。

Java 是一种“面向对象”的语言，这意味着它用对象来表示数据并提供与数据相关的方法。这种组织程序的方式是一种功能强大的设计理念，我们将在本书余下的篇幅中简单地介绍。

本章将介绍两种新的对象类型：`Point` 和 `Rectangle`；演示如何编写将对象作为参数的方法以及将对象作为返回值的方法；还将大致探讨一下 Java 类库的源代码。

10.1 Point对象

`java.awt` 包提供了一个名为 `Point` 的类，该类用于表示笛卡尔平面中的位置坐标。数学中通常将表示点的坐标放在括号内并用逗号分隔，例如，(0, 0) 表示原点，而 (x, y) 表示的点位于原点右方 x 个单位、上方 y 个单位。

要使用 `Point` 类的话必须先导入：

```
import java.awt.Point;
```

然后就可创建新的 `Point` 对象了。为此，必须使用运算符 `new`：

```
Point blank;
blank = new Point(3, 4);
```

第 1 行将 blank 的类型声明为 Point；第 2 行新建了一个以指定实参为坐标的 Point 对象。

运算符 new 的结果为指向新对象的引用，因此 blank 包含一个指向新 Point 对象的引用，如图 10-1 所示。

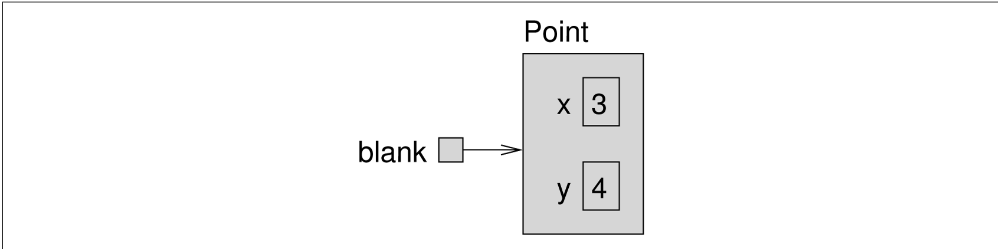


图 10-1：指向一个 Point 对象的变量的状态图

与往常一样，变量名 blank 位于方框外，其指向的对象的值位于方框内。变量 blank 的值在这里是一个引用，用箭头表示。这个箭头指向新对象，后者包含两个变量——x 和 y。

10.2　属性

属于对象的变量通常称为*属性*（attribute），也有人称之为“字段”。Java 用*句点表示法*（dot notation）访问对象的属性，如下所示：

```
int x = blank.x;
```

表达式 blank.x 的意思是，进入 blank 指向的对象，并获取其中的属性 x 的值。在这里，我们将这个值赋给了局部变量 x。局部变量 x 和属性 x 并不会发生冲突。句点表示法让你能够明确地指出引用的是哪个变量。

可以在表达式中使用句点表示法，如下所示：

```
System.out.println(blank.x + ", " + blank.y);
int sum = blank.x * blank.x + blank.y * blank.y;
```

第 1 行显示 3，4，第 2 行计算得到的值为 25。

10.3　将对象用作参数

可像通常那样将对象作为参数进行传递，例如：

```
public static void printPoint(Point p) {
    System.out.println("(" + p.x + ", " + p.y + ")");
}
```

这个方法接受一个 Point 参数，并在括号中显示其属性。如果调用 printPoint(blank)，它将显示 (3, 4)。

然而，我们并不需要 printPoint 这样的方法，因为调用 System.out.println(blank) 将得到如下输出：

```
java.awt.Point[x=3,y=4]
```

Point 对象提供了一个名为 toString 的方法，该方法返回点的字符串表示。以对象为参数调用 println 时，将自动调用 toString 并显示结果。这里显示的是类型名（java.awt.Point）以及属性的名称和值。

再来看一个例子。可将 6.2 节中的方法 distance 重写成将两个 Point 对象而不是四个 double 值作为参数：

```
public static double distance(Point p1, Point p2) {
    int dx = p2.x - p1.x;
    int dy = p2.y - p1.y;
    return Math.sqrt(dx * dx + dy * dy);
}
```

将对象作为参数可让源代码更易理解且不易出错，因为相关的值被关联起来了。

10.4 将对象作为返回类型

java.awt 包还提供了一个名为 Rectangle 的类。要使用这个类的话必须先导入：

```
import java.awt.Rectangle;
```

Rectangle 对象类似于 Point，但有四个属性：x、y、width 和 height。下面的示例创建了一个 Rectangle 对象，并让变量 box 指向它：

```
Rectangle box = new Rectangle(0, 0, 100, 200);
```

图 10-2 说明了这条赋值语句的作用。

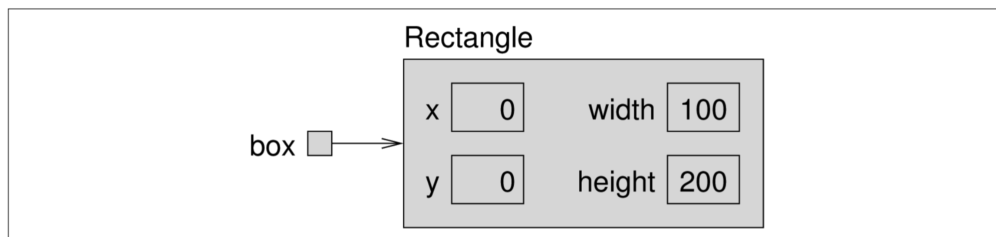


图 10-2：显示一个 Rectangle 对象的状态图

如果运行 System.out.println(box)，那么将得到如下输出：

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

同样，println 调用了 Rectangle 提供的方法 toString，这个方法知道如何显示 Rectangle 对象。

你可以编写返回对象的方法。例如，findCenter 接受一个 Rectangle 参数，并返回一个 Point，其中包含该矩形中心的坐标：

```
public static Point findCenter(Rectangle box) {
    int x = box.x + box.width / 2;
    int y = box.y + box.height / 2;
    return new Point(x, y);
}
```

这个方法的返回类型为 Point。最后一行创建了一个新的 Point 对象，并返回一个指向该对象的引用。

10.5　可修改的对象

可通过给对象的属性赋值来修改对象的内容。例如，要移动矩形而不改变其尺寸，可修改属性 x 和 y：

```
Rectangle box = new Rectangle(0, 0, 100, 200);
box.x = box.x + 50;
box.y = box.y + 100;
```

结果如图 10-3 所示。

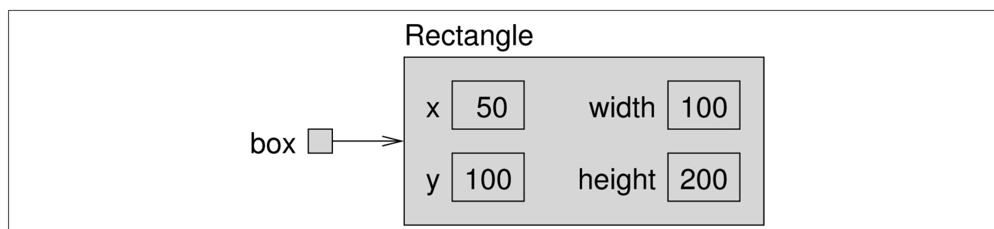


图 10-3：显示最新属性值的状态图

我们可将这些代码封装到一个方法中后再进行泛化，让这个方法能够将矩形移动任何指定的距离：

```
public static void moveRect(Rectangle box, int dx, int dy) {
    box.x = box.x + dx;
    box.y = box.y + dy;
}
```

变量 dx 和 dy 用于指定将矩形沿水平和垂直方向分别移多远。调用这个方法将修改作为实参传入的 Rectangle 对象。

```
Rectangle box = new Rectangle(0, 0, 100, 200);
moveRect(box, 50, 100);
System.out.println(box);
```

将对象作为实参传递给方法以便修改十分有用，但也可能导致调试更加困难，因为并非在任何情况下都能清楚地知道哪些方法会修改其实参。

Java 提供了很多操作 Point 和 Rectangle 对象的方法，例如，translate 的效果与 moveRect 相同，但调用这个方法时，不是将 Rectangle 对象作为实参传递给它，而是用句点表示法：

```
box.translate(50, 100);
```

这行代码对 box 指向的对象调用方法 translate，因此将直接更新对象 box。

这个示例很好地演示了面向对象（object-oriented）编程：不是编写 moveRect 这样修改一个或多个实参的方法，而是用句点表示法对对象调用方法。

10.6 指定别名

别忘了，将对象赋给变量时，赋给变量的实际上是指向对象的引用。可让多个变量指向同一个对象，如下所示：

```
Rectangle box1 = new Rectangle(0, 0, 100, 200);
Rectangle box2 = box1;
```

结果如图 10-4 所示。

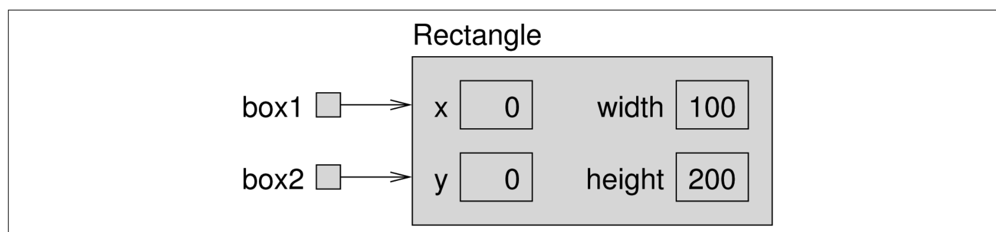


图 10-4：两个变量指向同一个对象的状态图

请注意，box1 和 box2 是同一个对象的别名，因此，影响其中一个变量的修改也将影响另一个变量。下面的示例将这个矩形的左上角向左、向上各移了 50 个单位，并将其高度和宽度都增加了 100 个单位：

```
System.out.println(box2.width);
box1.grow(50, 50);
System.out.println(box2.width);
```

第 1 行显示 `100`，这是 `box2` 指向的 `Rectangle` 对象的宽度；第 2 行对 `box1` 调用方法 `grow`，这个方法增大 `Rectangle` 对象的水平和垂直尺寸，结果如图 10-5 所示。

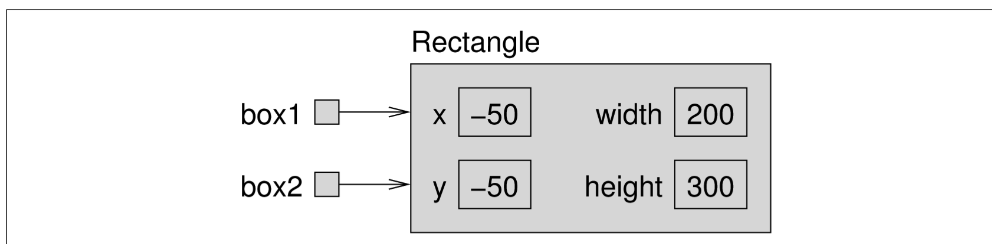


图 10-5：说明调用 `grow` 效果的状态图

通过 `box1` 所做的修改也将影响 `box2`，因此第 3 行显示的值为 `200`——增大后的矩形的宽度。

10.7　关键字null

创建对象变量时，别忘了你在这种变量中存储的是指向对象的引用。Java 中的关键字 null 是一个特殊值，意思是“没有指向任何对象”。可像下面这样声明并初始化对象变量：

```
Point blank = null;
```

状态图中用不带箭头的方框表示值 `null`，如图 10-6 所示。



图 10-6：包含 `null` 引用的对象变量的状态图

如果试图通过访问属性或调用方法来使用 `null` 值，那么将引发 `NullPointerException` 异常。

```
Point blank = null;
int x = blank.x;              // NullPointerException
blank.translate(50, 50);      // NullPointerException
```

另一方面，将 `null` 引用作为实参或返回值是完全合法的。例如，`null` 常用来表示特殊情况或指出错误。

10.8　垃圾收集

10.6 节说明了多个变量指向同一个对象会带来的后果。如果一个对象没有被任何变量指向，结果将如何呢？

```
Point blank = new Point(3, 4);
blank = null;
```

第 1 行创建了一个新的 `Point` 对象，并让 `blank` 指向它；第 2 行修改变量 `blank`，使其不指向任何对象。在状态图中，变量 `blank` 和这个 `Point` 对象之间的箭头将被删除，如图 10-7 所示。

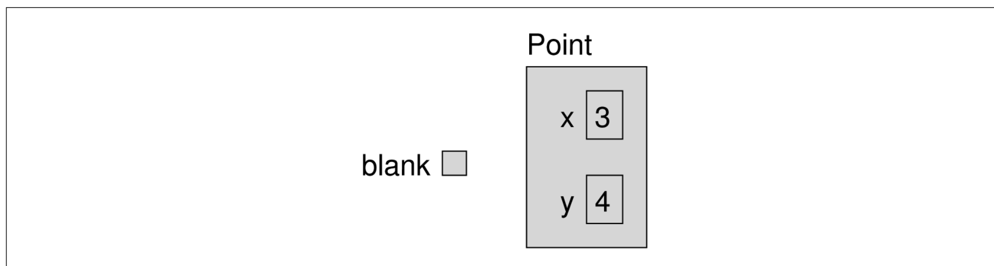


图 10-7：将变量设置为 `null` 效果的状态图

对于未被任何变量指向的对象，则无法访问其属性，也无法对其调用方法。在程序员看来，这样的对象已不复存在，但它依然驻留在计算机内存中，占据着内存空间。

系统会在程序运行时自动查找并回收无主对象；再将它们占据的空间用于存储新对象。这个过程被称为*垃圾收集*（garbage collection）。

垃圾收集是自动进行的，你什么都不用做，一般都意识不到垃圾收集过程的存在。但在高性能应用程序中，你可能时不时地会注意到些微的延迟，这是因为 Java 在回收被丢弃的对象所占据的空间。

10.9　类图

现在总结一下本章前面介绍的知识。`Point` 和 `Rectangle` 对象都有属性和方法；属性是对象的*数据*，方法是对象的*代码*；对象有哪些属性和方法由其所属的类定义。

在实践中，查看描述类的简图比研究其源代码更方便。*统一建模语言*（Unified Modeling Language，UML）定义了一种概述类设计的标准方式。

如图 10-8 所示，*类图*（class diagram）由两部分组成，上半部分列出了属性，下半部分列出了方法。UML 使用了一种独立于语言的格式，因此在类图中显示的不是 `int x`，而是 `x: int`。

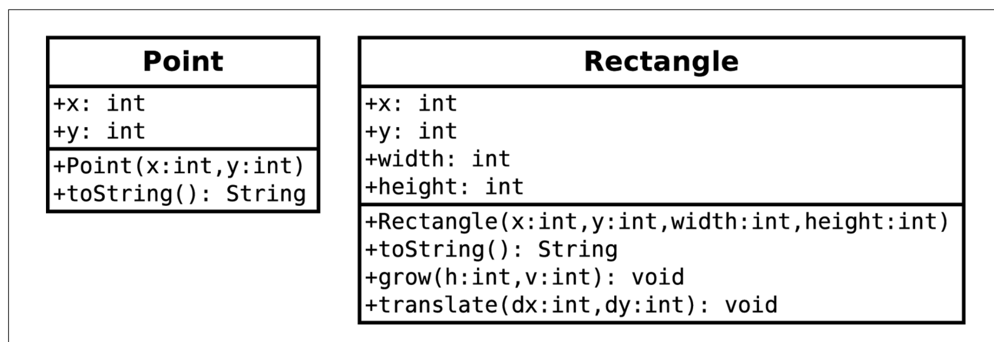


图 10-8：`Point` 和 `Rectangle` 的 URM 类图

状态图描述的是程序运行时的对象和变量，而类图描述的是编译时的源代码。

图 10-8 所示的类图中只列出了本章介绍过的方法，但 `Point` 和 `Rectangle` 都还包含其他方法。要想更详细地了解这些类的功能，请参阅相关的文档。

10.10　Java类库的源代码

本书一直在使用 Java 类库中的类，包括 `System`、`String`、`Scanner`、`Math`、`Random` 等。你可能还没有意识到这些类也是用 Java 编写的。事实上，可通过查看源代码来了解它们的工作原理。

Java 类库包含数千个文件，其中的很多文件都包含数千行代码。完全阅读并理解这些代码超出了个人的能力范围，因此千万不要感到害怕！

因为 Java 类库很大，所以其源代码存储在一个名为 src.zip 的文件中。请花点时间在你的计算机中找到这个文件。

- 在 Linux 系统中，它很可能位于目录 /usr/lib/jvm/openjdk-8/ 中（可能需要安装 `openjdk-8-source` 包）。
- 在 OS X 系统中，它很可能位于目录 /Library/Java/JavaVirtualMachines/jdk.../ Contents/Home/ 中。
- 在 Windows 系统中，它很可能位于目录 C:\Program Files\Java\jdk...\ 中。

将这个文件打开或解压后，你将看到与各个 Java 包对应的文件夹。例如，如果依次打开文件夹 `java` 和 `awt`，你将看到 `Point.java` 和 `Rectangle.java`，以及 `java.awt` 包中的其他类。

请在编辑器中打开 `Point.java`，并粗略地浏览一下这个文件。它使用了我们还未讨论的语言功能，因此可能很难理解，但通过浏览这个库，你可以感受一下专业的 Java 软件是什么样的。

注意，`Point.java` 包含详尽的文档。每个方法都有详尽的注释，包括 `@param`、`@return` 和其他 Javadoc 标签。Javadoc 通过阅读这些注释来生成 HTML 格式的文档。要想知道最终生成的 HTML 文件是什么样的，请阅读 `Point` 类的文档，而要找到这些文档，可在网上搜索 Java Point。

现在来看看 `Rectangle` 的方法 `grow` 和 `translate`。这些方法的功能比你所知道的要多，但这并不妨碍你在程序中使用它们。

这里对本章的内容作个总结。对象封装了数据并提供了可直接访问和修改这些数据的方法；面向对象编程能够将繁杂的细节隐藏起来，从而让你更轻松地使用和理解他人编写的代码。

10.11　术语表

- *属性*
  对象中的命名数据项。

- *句点表示法*
  用句点运算符（`.`）访问对象的属性或方法。

- *面向对象*
  将代码和数据组织成对象，而不是独立的方法。

- *垃圾收集*
  找出未被引用的对象并收回其占据的存储空间的过程。

- UML
  统一建模语言，软件工程领域中的一种标准绘图方式。

- *类图*
  对类的属性和方法进行描述的插图。

10.12　练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch10 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习10-1

这个练习旨在确保你明白将对象作为参数进行传递的机制。

(1) 为下面的程序绘制一个栈图，`riddle` 返回前，对方法 `main` 和 `riddle` 中的局部变量和形参进行描述。用箭头指出每个变量指向的对象。
(2) 这个程序的输出是什么？
(3) 对象 `blank` 是可修改的还是不可修改的？为什么？

```
public static int riddle(int x, Point p) {
    x = x + 7;
    return x + p.x + p.y;
}

public static void main(String[] args) {
    int x = 5;
    Point blank = new Point(1, 2);

    System.out.println(riddle(x, blank));
    System.out.println(x);
    System.out.println(blank.x);
    System.out.println(blank.y);
}
```

练习10-2

这个练习旨在确保你明白从方法返回对象的机制。

(1) 绘制一个栈图，指出下面这个程序在 `distance` 返回前的状态。列出这个栈图中的所有变量和形参，以及这些变量指向的对象。
(2) 这个程序的输出是什么？你能在不运行它的情况下确定这一点吗？

```
public static double distance(Point p1, Point p2) {
    int dx = p2.x - p1.x;
    int dy = p2.y - p1.y;
    return Math.sqrt(dx * dx + dy * dy);
}

public static Point findCenter(Rectangle box) {
    int x = box.x + box.width / 2;
    int y = box.y + box.height / 2;
    return new Point(x, y);
}

public static void main(String[] args) {
    Point blank = new Point(5, 8);

    Rectangle rect = new Rectangle(0, 2, 4, 4);
    Point center = findCenter(rect);

    double dist = distance(center, blank);
    System.out.println(dist);
}
```

练习10-3

这个练习与别名有关。前面说过，别名指的是两个指向同一个对象的变量。

(1) 绘制一个栈图，以显示下面程序在 `main` 结束前的状态，包括所有的局部变量及其指向的对象。
(2) 这个程序的输出是什么？
(3) `main` 结束时，`p1` 和 `p2` 互为别名吗？为什么？

```
public static void printPoint(Point p) {
    System.out.println("(" + p.x + ", " + p.y + ")");
}

public static Point findCenter(Rectangle box) {
    int x = box.x + box.width / 2;
    int y = box.y + box.height / 2;
    return new Point(x, y);
}

public static void main(String[] args) {
    Rectangle box1 = new Rectangle(2, 4, 7, 9);
    Point p1 = findCenter(box1);
    printPoint(p1);

    box1.grow(1, 1);
    Point p2 = findCenter(box1);
    printPoint(p2);
}
```

练习10-4

你可能对 `factorial` 方法感到厌烦了，但你还得再编写一个版本。

(1) 新建一个命名为 Big.java 的程序，并编写（或重用）方法 `factorial` 的迭代版本。
(2) 以表格的方式显示整数 0~30 及其阶乘。你可能会发现，结果到 15 左右就不再正确了。这是为什么呢？
(3) `BigInteger` 是一个 Java 类，可用于表示任意大的整数，它可表示的最大整数只受制于内存量和处理速度。请花点时间阅读这个类的文档。可在网上搜索 Java BigInteger 来找这个文档。
(4) 要想使用 `BigInteger`，必须在程序开头导入 `java.math.BigInteger`。
(5) 创建 `BigInteger` 对象的方式有很多种，但最简单的方式是使用 `valueOf`。下面的代码将一个整数转换为 `BigInteger` 对象：

```
int x = 17;
BigInteger big = BigInteger.valueOf(x);
```

(6) 因为 `BigInteger` 不是基本数据类型，所以对其执行数学运算时，不能用常规的数学运算符，而必须用 `add` 等方法。要想将两个 `BigInteger` 相加，可对其中一个调用方法

add，并将另一个作为实参：

```
BigInteger small = BigInteger.valueOf(17);
BigInteger big = BigInteger.valueOf(1700000000);
BigInteger total = small.add(big);
```

请尝试使用其他方法，如 multiply 和 pow。

(7) 修改方法 factorial，在其中用 BigInteger 来执行计算，并将结果作为 BigInteger 返回。可保留其中的形参，因为其类型还是 int。
(8) 修改方法 factorial 后，再次尝试以表格的方式显示 0~30 及其阶乘。计算出的 30 的阶乘正确吗？参数值最多可到多少结果依然是正确的？
(9) BigInteger 对象是可修改的还是不可修改的？你是怎么知道的？

练习10-5

很多加密算法需要计算大整数的幂，下面的方法实现了一种高效的整数幂算法：

```
public static int pow(int x, int n) {
    if (n == 0) return 1;

    // 递归地计算x的n/2次幂
    int t = pow(x, n / 2);

    // 如果n为偶数,结果就是t的平方
    // 如果n为奇数,结果就是t的平方乘以x
    if (n % 2 == 0) {
        return t * t;
    } else {
        return t * t * x;
    }
}
```

这个方法存在的问题是，仅当结果小到能够用 int 类型表示时，它才管用。请修改这个方法，在其中用 BigInteger 对象来存储结果，但形参可保留为 int 类型。

你应该使用 BigInteger 的方法 add 和 multiply，但不要使用方法 BigInteger.pow，不然就太没意思了。

第11章

类

每当定义新类时，就创建了一个同名的新类型。因此，在 1.4 节中定义 `Hello` 类时，就创建了一种名为 `Hello` 的类型。我们没有声明任何类型为 `Hello` 的变量，也没有用 `new` 创建 `Hello` 对象。即便这样做了，创建出的 `Hello` 对象的用处也不大，但我们确实可以这样做！

我们将在本章中定义表示有用的对象类型的类，还将表明类和对象之间的差别。下面列出了一些最重要的理念。

- 定义类（class）就创建了同名的对象类型。
- 每个对象都属于某种对象类型，即是某个类的*实例*（instance）。
- 类定义相当于创建对象的模板，指定了对象包含哪些属性以及哪些方法可以操作这些属性。
- 类犹如建筑设计图，可根据同一张设计图建造出很多房子。
- 操作对象的方法是在对象所属的类中定义的。

11.1　Time类

为何要定义新类呢？这样做的一个常见目的是将相关的数据封装到对象中，以便能够将它们视为一个整体。这样我们就可以将对象用作参数和返回值，而不传递和返回多个值。这种设计原则被称为*数据封装*（data encapsulation）。

前面介绍了两个以这种方式封装数据的类型：`Point` 和 `Rectangle`。另一个这样的类型是表示时间的 `Time`，我们将在本章实现它。`Time` 对象封装的数据为小时、分钟和秒数。因为每

个 `Time` 对象都包含这些数据，所以我们将定义存储它们的属性。

属性也被称为*实例变量*（instance variable），因为每个实例都包含这些变量，与之相反的是类变量（将在 12.3 节中介绍）。

第一步是判断各个变量应为什么类型。`hour` 和 `minute` 显然应为整数，但为了让这个类有趣些，我们将 `second` 声明为 `double` 类型。

实例变量是在类定义开头（而不是方法中）声明的。下述代码片段本身就是一个合法的类定义：

```
public class Time {
    private int hour;
    private int minute;
    private double second;
}
```

`Time` 类是公有的，这意味着可在其他类中使用。但这些实例变量是私有的，这意味着只能在 `Time` 类中访问。如果你试图在其他类中读写它们，那么将导致编译错误。

将实例变量声明为私有的有助于将类隔离开来，避免修改一个类后必须相应地修改其他类；还可让其他程序员在使用你编写的类时，减少需要明白的内容。这种隔离称为*信息隐藏*（information hiding）。

11.2 构造函数

声明实例变量后的下一步是定义*构造函数*（contructor）——初始化实例变量的特殊方法。构造函数的定义语法与其他方法类似，但：

- 构造函数与类同名；
- 构造函数没有返回类型（因此没有返回值）；
- 不使用关键字 `static`。

下面是 `Time` 类的一个构造函数：

```
public Time() {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

这个构造函数不接受任何实参，其中的每行代码都将一个实例变量初始化为零（就这里而言，这意味着午夜）。

`this` 是一个关键词，指向正在创建的对象。可像使用对象名一样使用 `this`。例如，可读写

this 的实例变量，将 this 作为实参传递给方法。然而，this 并不是你声明的，不能给它赋值。

编写构造函数时的常见错误是在末尾添加一条 return 语句。与 void 方法一样，构造函数不返回值。

要想创建 Time 对象，必须使用运算符 new：

```
Time time = new Time();
```

调用 new 时，Java 将创建指定的对象，并调用构造函数来初始化其实例变量。构造函数执行完毕后，new 将返回一个指向新对象的引用。在这个示例中，引用被赋给了类型为 Time 的变量 time，结果如图 11-1 所示。

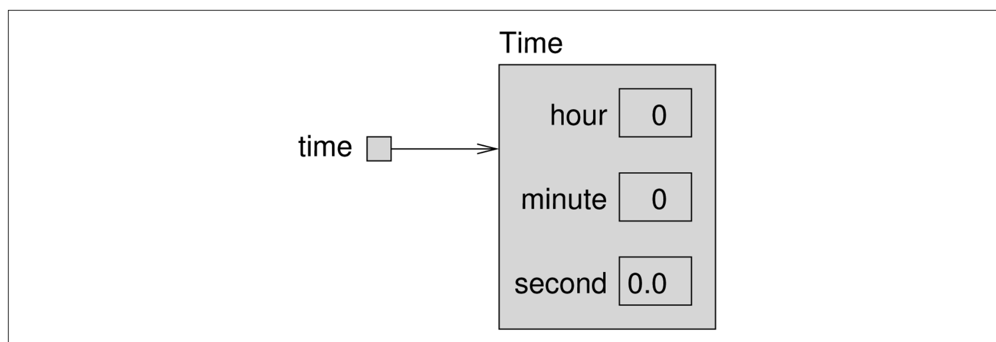


图 11-1：Time 对象的状态图

初学者有时会犯这样的错误，即在构造函数中调用 new。不必这样做，也不能这样做。在这个示例中，在构造函数中调用 new Time() 将导致无限递归：

```
public Time() {
    new Time();          // 不对!
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

11.3　再谈构造函数

与其他方法一样，构造函数也可重载，这意味着可提供形参不同的多个构造函数。Java 知道该调用哪个构造函数，这是根据你提供的实参确定的。

一种常见的做法是，在提供一个不接受任何参数的构造函数（如前面的构造函数）的同时，提供一个“值构造函数”，如下：

```
public Time(int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
```

这个构造函数所做的只是将形参的值复制到实例变量中。在这个实例中，形参的名称和类型与实例变量相同，因此形参将*遮盖*（shadow）或者说隐藏实例变量。为将它们区分开来，关键词 `this` 必不可少。让形参与相应的实例变量同名是一种常见的做法，但并非必须这样。

要调用这个构造函数，必须在运算符 `new` 后面提供实参。下面的示例创建了一个 `Time` 对象，这个对象表示的时间为正午前 0.1 秒：

```
Time time = new Time(11, 59, 59.9);
```

重载构造函数可提供这样的灵活性，即先创建对象再填充属性或在创建对象本身前收集所有的信息。

一旦掌握构造函数的编写技巧，你就会觉得这样的工作很乏味。只需看一眼实例变量列表就能快速编写出构造函数。事实上，有些 IDE 都能替你生成构造函数。

将前面的代码整合起来，得到如下完整的类定义：

```
public class Time {
    private int hour;
    private int minute;
    private double second;

    public Time() {
        this.hour = 0;
        this.minute = 0;
        this.second = 0.0;
    }

    public Time(int hour, int minute, double second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
}
```

11.4 获取方法和设置方法

`Time` 类的实例变量是私有的，可在 `Time` 类中访问，但如果试图在其他类中访问它们，那么编译器将报错。

例如，下面是一个名为 TimeClient 的新类，之所以这样命名，是因为使用其他类定义的对象的类称为*客户端*（client）：

```
public class TimeClient {

    public static void main(String[] args) {
        Time time = new Time(11, 59, 59.9);
        System.out.println(time.hour); // compiler error
    }
}
```

如果你尝试编译这些代码，将出现类似于下面这样的错误消息：hour has private access in Time（Time 类中的 hour 是私有的）。解决这种问题的办法有三种。

- 将实例变量 hour 声明为公有的。
- 在 Time 类中提供访问其实例变量的方法。
- 因为本来就要禁止其他类访问 Time 类的实例变量，所以这根本就不是问题。

第一种选择很有吸引力，因为它简单易行，但问题是如果类 A 直接访问类 B 的实例变量，那么类 A 将“依赖”于类 B。换而言之，每当你修改类 B 时，很可能也必须修改类 A。

如果类 A 只使用类 B 的方法来与类 B 交互，那么它们将是彼此“独立的”，这意味着修改类 B 不会影响类 A（只要类 A 使用的方法的特征标没有变化）。

因此，如果我们认为必须让 TimeClient 能够读取 Time 的实例变量，可在 Time 中提供执行这种任务的方法：

```
public int getHour() {
    return this.hour;
}

public int getMinute() {
    return this.minute;
}

public int getSecond() {
    return this.second;
}
```

这种方法的正规名称是“访问器”，但更常用的名称是*获取方法*（getter）。根据约定，获取实例变量 something 的方法将被命名为 getSomething。

如果我们认为还必须让 TimeClient 能够修改 Time 的实例变量，也可在 Time 中提供执行这种任务的方法：

```
public void setHour(int hour) {
    this.hour = hour;
}
```

```
    public void setMinute(int minute) {
        this.minute = minute;
    }

    public void setSecond(int second) {
        this.second = second;
    }
```

这些方法的正规名称是“修改器”，但更常用的叫法是*设置方法*（setter）。设置方法的命名约定与获取方法类似：设置实例变量 `something` 的方法将被命名为 `setSomething`。

编写获取方法和设置方法的工作可能很繁琐，但很多 IDE 都能够根据实例变量生成这些方法。

11.5 显示对象

如果创建一个 `Time` 对象，并用 `println` 显示：

```
    public static void main(String[] args) {
        Time time = new Time(11, 59, 59.9);
        System.out.println(time);
    }
```

输出将类似于下面这样：

```
    Time@80cc7c0
```

当要求显示对象类型的值时，Java 显示类型名和对象的地址（十六进制表示）。如果需要跟踪各个对象，这种地址在调试中很有用。

要想以对用户更有意义的方式来显示 `Time` 对象，可编写一个显示小时、分钟和秒的方法。为此，可以 4.6 节中的方法 `printTime` 为基础，编写下面的方法：

```
    public static void printTime(Time t) {
        System.out.print(t.hour);
        System.out.print(":");
        System.out.println(t.minute);
        System.out.print(":");
        System.out.println(t.second);
    }
```

如果用前一节的 `time` 对象来调用这个方法，输出将为 `11:59:59.9`。为让这个方法的代码更简洁，可用 `printf`：

```
    public static void printTime(Time t) {
        System.out.printf("%02d:%02d:%04.1f\n",
            t.hour, t.minute, t.second);
    }
```

需要提醒你的是，整数需要使用格式说明符 `%d`，浮点数需要使用格式说明符 `%f`。选项 `02` 表示总共两位，如果不够就在前面添加零；选项 `04.1` 表示整数部分 4 位、小数部分 1 位，如果不够就在前面添加零。

11.6 方法toString

每种对象类型都有一个名为 `toString` 的方法，用于返回对象的字符串表示。用 `print` 或 `println` 显示对象时，Java 将调用其方法 `toString`。

这个方法默认显示对象的类型和地址，但可通过提供方法 `toString` 来*覆盖*（override）这种行为。例如，下面是 `Time` 的方法 `toString`：

```
public String toString() {
    return String.format("%02d:%02d:%04.1f\n",
        this.hour, this.minute, this.second);
}
```

这个方法的定义没有包含关键字 `static`，因为它不是静态方法，而是*实例方法*（instance method）。为何叫实例方法呢？因为必须通过类（这里是 `Time`）的实例来调用。实例方法有时也被称为非静态方法，你可能在错误消息中见过这个术语。

这个方法的方法体与前一节的 `printTime` 类似，但有两个地方不同：

- 在这个方法中，我们使用了 `this` 来引用当前实例，即对其调用该方法的对象；
- 使用的不是 `printf` 而是 `String.format`；`String.format` 返回一个指定格式的字符串，而不是显示它。

现在可以直接调用 `toString` 了：

```
Time time = new Time(11, 59, 59.9);
String s = time.toString();
```

还可通过 `println` 间接地调用它：

```
System.out.println(time);
```

在这个示例中，`toString` 中的 `this` 与 `time` 指的是同一个对象。输出为 `11:59:59.9`。

11.7 方法equals

本书前面介绍了两种检查两个值是否相等的方式：运算符 `==` 和方法 `equals`。这两种方式都可用于对象，但它们的含义不同。

- 运算符 `==` 检查两个对象是否*相同*（identical），即指的是否是同一个对象。

- 方法 `equals` 检查两个对象是否相等（equivalent），即它们的值是否相同。

相同的定义是固定不变的，因此运算符 == 所做的事情始终相同；但相等的定义随对象而异，因此对象可定义自己的方法 `equals`。 请看下面的变量：

```
Time time1 = new Time(9, 30, 0.0);
Time time2 = time1;
Time time3 = new Time(9, 30, 0.0);
```

图 11-2 所示的状态图显示了这些变量及其值。

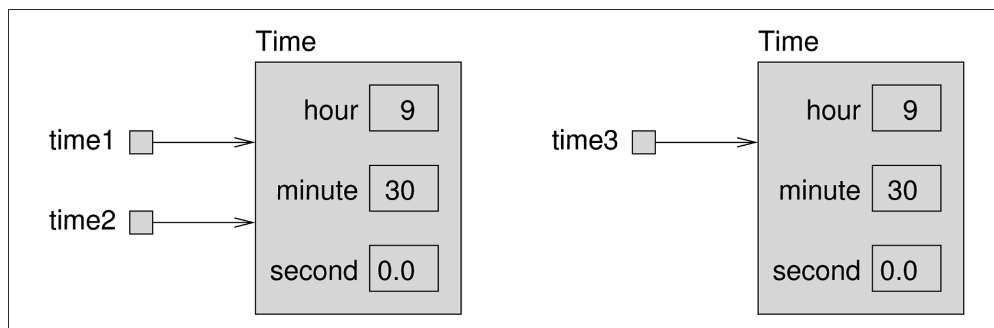


图 11-2：显示三个 `Time` 变量的状态图

赋值运算符复制引用，因此 `time1` 和 `time2` 指向同一个对象。因为它们相同，所以 `time1 == time2` 为 `true`。

然而，`time1` 和 `time3` 指向不同的对象。因为它们不同，所以 `time1 == time3` 为 `false`。

默认情况下，方法 `equals` 的行为与 == 相同。就 `Time` 对象而言，这可能不是我们所希望的。例如，`time1` 和 `time3` 表示的时间相同，因此，应将它们视为相等。

我们可提供一个实现这种相等定义的 `equals` 方法：

```
public boolean equals(Time that) {
    return this.hour == that.hour
        && this.minute == that.minute
        && this.second == that.second;
}
```

`equals` 是一个实例方法，因此，它使用 `this` 来引用当前对象，且其定义中没有包含关键词 `static`。我们可以像下面这样来调用这个 `equals` 方法：

```
time1.equals(time3);
```

在这个 `equals` 方法中，`this` 指的是对象 `time1`，而 `that` 指的是对象 `time3`。因为这两个对象的实例变量相等，所以结果为 `true`。

很多对象采用了类似的相等定义，即如果两个对象的实例变量相等，那么这两个对象就相等。然而，也可用其他的方式定义相等。

11.8 时间相加

假设你要去看电影，影片开始放映的时间为 18:50（6:50PM），时长为 2 小时 16 分钟，请问影片将在什么时候结束？

我们用 `Time` 对象来执行这种计算。可采取下面两种方式将 Time 对象“相加”：

- 编写一个将两个 `Time` 对象作为参数的静态方法；
- 编写一个通过 `Time` 对象进行调用，并将一个 Time 对象作为参数的实例方法。

为说明差别，我们将演示这两种方式。静态方法的代码类似于下面这样：

```
public static Time add(Time t1, Time t2) {
    Time sum = new Time();
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
    sum.second = t1.second + t2.second;
    return sum;
}
```

下面的代码演示了如何调用这个静态方法：

```
Time startTime = new Time(18, 50, 0.0);
Time runningTime = new Time(2, 16, 0.0);
Time endTime = Time.add(startTime, runningTime);
```

另一方面，实例方法的代码类似于下面这样：

```
public Time add(Time t2) {
    Time sum = new Time();
    sum.hour = this.hour + t2.hour;
    sum.minute = this.minute + t2.minute;
    sum.second = this.second + t2.second;
    return sum;
}
```

所做的修改如下。

- 删除了关键词 `static`。
- 删除了第一个形参。
- 将 `t1` 替换成了 `this`。

还可以将 `t2` 替换为 `that`。不同于 `this`，`that` 并非关键词，而只是一个比 `t2` 更合适的变量名。

下面的代码演示了如何调用这个实例方法：

```
Time endTime = startTime.add(runningTime);
```

这就是将时间相加的静态方法和实例方法。静态方法和实例方法的功能相同，要想在它们之间转换，只需要修改几个地方。

然而，还有一个问题，那就是执行加法的代码不正确。就这个示例而言，返回的结果为 `20:66`，这并非有效的时间。如果 `second` 超过了 59，那么就必须进位到 `minute`，而如果 `minute` 超过了 59，就必须进位到 `hour`。

下面是一个更佳的 `add` 版本：

```
public Time add(Time t2) {
    Time sum = new Time();
    sum.hour = this.hour + t2.hour;
    sum.minute = this.minute + t2.minute;
    sum.second = this.second + t2.second;

    if (sum.second >= 60.0) {
        sum.second -= 60.0;
        sum.minute += 1;
    }
    if (sum.minute >= 60) {
        sum.minute -= 60;
        sum.hour += 1;
    }
    return sum;
}
```

然而，`hour` 也可能超过 23，但 `Time` 类没有属性 `days`，因此无法进位。在 `hour` 超过 23 时，可用 `sum.hour -= 24` 来获得正确的结果。

11.9 纯方法和非纯方法

前面的静态方法 `add` 没有修改任何形参，而是创建并返回了一个新的 `Time` 对象。作为一种替代方案，可编写一个类似于下面这样的方法：

```
public void increment(double seconds) {
    this.second += seconds;
    while (this.second >= 60.0) {
        this.second -= 60.0;
        this.minute += 1;
    }
    while (this.minute >= 60) {
        this.minute -= 60;
        this.hour += 1;
    }
}
```

方法 `increment` 直接修改当前的 `Time` 对象，而不是创建并返回一个新的 `Time` 对象。

像 `add` 那样的方法被称为*纯方法*（pure method），因为：

- 它们没有修改形参；
- 它们没有任何“副作用”，如打印；
- 它们的返回值完全取决于形参，而不受任何其他状态的影响。

方法 `increment` 违反了第一条规则，像它这样的方法有时被称为*非纯方法*（modifier）。非纯方法通常是 `void` 方法，但有些也返回一个引用，用于指向它们修改的对象。

因为非纯方法不创建对象，所以效率可能更高，但也更容易出错。如果对象被多个变量指向，可能难以搞清楚非纯方法带来的影响。

要想让类像 `String` 那样不可修改，可以只提供获取方法，而不提供设置方法，同时只提供纯方法，而不提供非纯方法。不可修改的对象看似用起来更麻烦，但可节省大量的调试时间。

11.10　术语表

- *类*
  在本书前面，我们将类定义为相关方法的集合。现在你应该知道类也是创建对象的模板。

- *实例*
  类的一员。每个对象都是某个类的实例。

- *数据封装*
  将多个命名变量放在单个对象中。

- *实例变量*
  对象的属性，即在类中定义的非静态变量。

- *信息隐藏*
  将实例变量声明为私有的，以减少类之间的依赖关系。

- *构造函数*
  对新创建的对象的实例变量进行初始化的特殊方法。

- *遮盖*
  定义类型和名称与实例变量相同的局部变量或形参。

- *客户端*
  使用其他类定义的对象的类。

- *获取方法*
  返回实例变量的值的方法。

- *设置方法*
  给实例变量赋值的方法。

- *覆盖*
  替换方法（如 `toString`）的默认实现。

- *实例方法*
  可访问 `this` 和实例变量的非静态方法。

- *相同*
  两个一样的值；就对象变量而言，指的是它们指向同一个对象。

- *相等*
  在方法 `equals` 看来，两个对象是相等的，但不一定相同。

- *纯方法*
  结果只取决于形参，而不受其他数据影响的静态方法。

- *非纯方法*
  修改对象状态（实例变量）的方法。

11.11 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch11 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

至此，你已经具备了足够的知识，应该能够看懂介绍简单 2D 图形和动画的附录 B。阅读接下来的几章时，你应抽空阅读这个附录并完成各章的练习。

练习11-1

阅读 `java.awt.Rectangle` 的文档，看看哪些方法是纯方法？哪些是非纯方法？

如果阅读 `java.lang.String` 的文档，你将发现它没有非纯方法，因为字符串是不可修改的。

练习11-2

本章中的 `increment` 方法的实现效率不太高，你能重写这个方法，使其不使用任何循环吗？提示：别忘了求模运算符。

练习11-3

在图版游戏 Scrabble 中，每个卡片都包含一个字母和一个分数，其中前者用于在行列上拼出单词，后者用于计算单词的价值。

(1) 请为表示卡片的 `Tile` 类编写定义。这个类应包含 `char` 实例变量 `letter` 和 `int` 实例变量 `value`。
(2) 编写一个构造函数，让它包含形参 `letter` 和 `value`，并初始化前述实例变量。
(3) 编写一个名为 `printTile` 的方法，它将一个 `Tile` 对象作为参数，并以方便阅读的格式显示该对象的实例变量。
(4) 编写一个名为 `testTile` 的方法，让它创建一个属性 `letter` 和 `value` 分别为 `Z` 和 `10` 的 `Tile` 对象，再用 `printTile` 来显示这个对象的状态。
(5) 为 `Tile` 类实现方法 `toString` 和 `equals`。
(6) 为每个属性创建获取方法和设置方法。

这个练习旨在让你熟悉创建类定义以及编写测试类的代码。

练习11-4

对象类型 `Date` 包含三个 `int` 实例变量：`year`、`month` 和 `day`，请为它编写类定义。这个类应提供两个构造函数，其中一个没有任何形参，并将实例变量初始化为默认值；另一个包含形参 `year`、`month` 和 `day`，并用它们来初始化实例变量。

编写一个 `main` 方法，让它创建一个表示你生日的 `Date` 对象——`birthday`。可用前述任何一个构造函数创建这个对象。

练习11-5

有理数是可表示为分数的数字。例如，2/3 就是一个有理数，数字 7 也是有理数，因为可将其视为 7/1。

(1) 定义一个名为 `Rational` 的类，让它包含两个 `int` 实例变量，分别用于存储分子和分母。
(2) 编写一个构造函数，让它不接受任何参数，并将分子和分母分别设置为 0 和 1。
(3) 编写一个名为 `printRational` 的实例方法，并以合理的格式显示 `Rational` 对象。
(4) 编写一个 `main` 方法，让它创建一个 `Rational` 对象、设置该对象的实例变量并显示该对象。
(5) 至此，你编写了一个最基本的程序，请对其进行测试，必要时进行调试。
(6) 为 `Rational` 类编写方法 `toString`，并用 `println` 对其进行测试。
(7) 再编写一个构造函数，让它包含两个形参，并用它们来初始化实例变量。
(8) 编写一个名为 `negate` 的实例方法，让它对有理数求负。这是一个非纯方法，因此也是

void 方法。在 main 中添加对这个方法进行测试的代码。

(9) 编写一个名为 invert 的实例方法，让它通过将分子和分母互换来计算倒数。这是一个非纯方法。在 main 中添加对这个方法进行测试的代码。

(10) 编写一个名为 toDouble 的实例方法，让它将有理数转换为 double 值（浮点数），并返回结果。这是一个纯方法，不修改对象。与前面一样，请对这个方法进行测试。

(11) 编写一个名为 reduce 的实例方法，让它将有理数化简为最简分数：找出分子和分母的最大公约数，并将分子和分母都除以这个公约数。这是一个纯方法，不修改当前对象的实例变量。

提示：找出最大公约数只需要几行代码。要了解这方面的更详细信息，请在网上搜索 Euclidean algorithm。

(12) 编写一个名为 add 的实例方法，让它将一个 Rational 对象作为参数，将其与 this 相加，并返回一个新的 Rational 对象。

将分数相加的方法有很多种。你可根据自己的喜好选择，但务必对结果进行化简，让分子和分母没有除 1 之外的其他公约数。

这个练习旨在让你编写包含各种方法的类定义：构造函数、静态方法、实例方法、非纯方法和纯方法。

第 12 章

对象数组

在余下的几章中，我们将开发处理单张扑克牌和整副扑克牌的程序。这里大致说一下后面要做的工作。

- 在本章中，我们将定义一个 `Card` 类，并编写处理单张扑克牌和扑克牌数组的方法。
- 在第 13 章中，我们将编写一个 `Deck` 类（封装了一个扑克牌数组），并编写操作整副牌的方法。
- 在第 14 章中，我们将介绍继承——一种通过扩展已有类来创建新类的方式，然后用所有这些类来实现扑克牌游戏 *Crazy Eights*。

本章的代码位于 Card.java 中，这个文件可在本书的代码仓库目录 ch12 中找到。有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。

12.1 Card对象

若你不熟悉扑克牌，现在去买一副并访问 https://en.wikipedia.org/wiki/Standard_52-card_deck 正当其时。

一副标准的扑克牌有 52 张，每张扑克牌都为 4 种花色（suit）和 13 个点数（rank）之一。四种花色为黑桃、红心、方块和梅花；13 个点数为 A、2、3、4、5、6、7、8、9、10、J、Q 和 K。

如果要定义一个表示单张扑克牌的类，这个类显然应包含如下实例变量：`rank` 和 `suit`，但这些实例变量应为什么类型却不那么明显。一种选择是将它们的类型声明为 `String`，这样

实例变量 suit 将包含 "Spade" 这样的字符串，而 rank 将包含 "Queen" 这样的字符串。这种设计存在的一个问题是，比较两张扑克牌的点数或花色将不那么容易。

另一种设计方案是用整数将点数和花色进行*编码*（encode），这里的“编码”指的并非加密（转换为看不懂的内容），而是在数字序列和要表示的东西之间建立映射。

下面是为花色建立的一种映射：

梅花 $\mapsto$ 0

方块 $\mapsto$ 1

红心 $\mapsto$ 2

黑桃 $\mapsto$ 3

这里使用了数学符号 $\mapsto$ ，旨在明确地指出这些映射并非程序的组成部分。它们是程序设计的一部分，但不会出现在代码中。

花牌与数字的映射关系如下，其他扑克牌对应其点数表示的数字（2~10）：

A $\mapsto$ 1

J $\mapsto$ 11

Q $\mapsto$ 12

K $\mapsto$ 13

至此，Card 类的定义类似于下面这样：

```
public class Card {
    private int rank;
    private int suit;

    public Card(int rank, int suit) {
        this.rank = rank;
        this.suit = suit;
    }
}
```

实例变量被声明为私有的：可以在这个类中访问，但不能在其他类中访问。

构造函数包含两个对应于实例变量的形参。可用运算符 new 创建 Card 对象：

```
Card threeOfClubs = new Card(3, 0);
```

结果为一个引用，指向表示梅花 3 的 Card 对象。

12.2 方法toString

创建新类的第一步是声明实例变量和编写构造函数。接下来该干什么呢？一种不错的选择是编写方法 toString，这对调试和渐进开发大有帮助。

要想以易于阅读的方式显示 Card 对象，需要将整数编码映射到字符串。为此，一种比较自然的方式是使用一个 String 数组。我们可以像下面这样创建这个数组：

```
String[] suits = new String[4];
```

然后给每个元素赋值：

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

我们也可像 8.3 节那样，在创建数组的同时初始化元素：

```
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
```

图 12-1 所示的状态图显示了结果。每个元素都是一个指向 String 的引用。

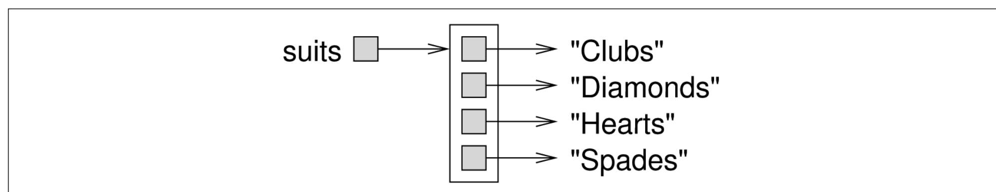


图 12-1：一个字符串数组的状态图

我们还需要一个对点数进行解码的数组：

```
String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",
           "7", "8", "9", "10", "Jack", "Queen", "King"};
```

有效的点数为 1~13，因此根本不会用到索引为 0 的元素。我们将它设置为 null，表明这个元素不会被用到。

通过将实例变量 suit 和 rank 用作索引，可从这些数组中提取有意义的字符串。

```
String s = ranks[card.rank] + " of " + suits[card.suit];
```

表达式 suits[card.suit] 表示将对象 card 的实例变量 suit 用作索引，以访问数组 suits 中的相应元素。

现在可以将这些代码封装到方法 toString 中了：

```
public String toString() {
    String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",
               "7", "8", "9", "10", "Jack", "Queen", "King"};
    String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
    String s = ranks[this.rank] + " of " + suits[this.suit];
    return s;
}
```

当我们显示 Card 对象时，println 将自动调用 toString：

```
Card card = new Card(11, 1);
System.out.println(card);
```

输出为 Jack of Diamonds。

12.3 类变量

到目前为止，你已经见过了局部变量和实例变量，其中局部变量是在方法中声明的，而实例变量是在类定义中声明的，通常位于类定义前面。

局部变量是在方法被调用时创建的，方法结束时，它们占据的内存空间将被收回。实例变量是在创建对象时创建的，在对象被作为垃圾收集时，它们占据的内存空间将被收回。

现在该介绍*类变量*（class variable）了。与实例变量一样，类变量也是在类定义中声明的，但使用了关键词 static 对其进行标识。它们创建于程序开始运行（或所属类首次被使用）时，直到程序结束才消失。类变量由其所属类的所有实例*共享*。

```
public class Card {

    public static final String[] RANKS = {
        null, "Ace", "2", "3", "4", "5", "6", "7",
        "8", "9", "10", "Jack", "Queen", "King"};

    public static final String[] SUITS = {
        "Clubs", "Diamonds", "Hearts", "Spades"};

    // 这里为实例变量和构造函数

    public String toString() {
        return RANKS[this.rank] + " of " + SUITS[this.suit];
    }
}
```

类变量常用于存储多个地方要用到的常量值。在这种情况下，还应将它们声明为 final。请注意，决定将变量声明为 static 或 final 时，需要考虑两个不同的因素：如果变量由所有实例共享，那么就将其声明为 static；如果变量为常量，就应将其声明为 final。

对于 static final 变量来说，一种常用的命名约定是采用全大写，这更容易让人知道它们在类中扮演的角色。在方法 toString 中，我们可以像引用局部变量一样引用 SUITS 和 RANKS，但要能够判断出它们是类变量。

将 SUITS 和 RANKS 定义为类变量的另一个优点是，无需在每次调用 toString 时创建它们，也不需要在这个方法执行完毕后将它们视为垃圾进行收集。其他方法和类也可能需要它们，因此让它们在任何地方都可用很有帮助。由于这些数组变量是 final 的，且其元素指向的字符串是不可修改的，因此将它们声明为公有的不会有任何危险。

12.4 方法compareTo

正如你在 11.7 节中看到的，创建测试两个对象是否相等的方法 equals 大有裨益。

```
public boolean equals(Card that) {
    return this.rank == that.rank
        && this.suit == that.suit;
}
```

另外，如果有比较两张扑克牌的方法就好了，这样就能知道哪张牌更大。 可用比较运算符（<、> 等）比较基本类型值，但这些运算符对对象类型不管用。

正如你在 9.6 节看到的，Java 类 String 提供了方法 compareTo。与方法 equals 一样，我们也可以为自定义类编写方法 compareTo 的定制版本。

有些类型是“完全有序的”，即可以通过比较判断出任何两个值的大小。整数和字符串都是完全有序的。

其他的类型是“无序的”，即无法以有意义的方式来规定哪个元素更大。在 Java 中，boolean 类型是无序的；如果你编写表达式 true < false，编译器将报错。

扑克牌是“部分有序的”，这意味着有些扑克牌能够比较，有些不能。例如，我们知道梅花 3 比梅花 2 大，方块 3 比梅花 3 大，但梅花 3 和方块 2 哪个更大呢？它们一个点数更大，另一个花色更大。

要想让扑克牌是可以比较的，必须指定哪个更重要：点数还是花色。如何选择没有定规，不同的游戏可能作出不同的选择。但刚买的扑克牌是有序的，开头全部是梅花，然后全部是方块，依次类推。因此，我们现在暂时假设花色更重要。

根据上述假设，可这样编写方法 compareTo：

```
public int compareTo(Card that) {
    if (this.suit < that.suit) {
        return -1;
    }
```

```
        if (this.suit > that.suit) {
            return 1;
        }
        if (this.rank < that.rank) {
            return -1;
        }
        if (this.rank > that.rank) {
            return 1;
        }
        return 0;
    }
```

如果 this 更大，compareTo 将返回 1；如果 that 更大，就返回−1；如果两者相等，就返回 0。先比较花色，如果花色相同，再比较点数，如果点数也相同，就返回 0。

12.5 Card对象是不可修改的

Card 的实例变量都是私有的，因此在其他类中无法访问。我们可提供获取方法，让其他类能够读取实例变量 rank 和 suit 的值：

```
    public int getRank() {
        return this.rank;
    }

    public int getSuit() {
        return this.suit;
    }
```

是否提供设置方法属于设计方面的决策。如果我们提供了，Card 对象将是可修改的，因此可以将一张牌变成另一张牌。这可能不是我们所需要的功能，而且一般而言，可修改的对象更容易出错。因此，让 Card 对象不可修改可能是更好的选择。为此，只要不提供任何非纯方法（包括设置方法）就可以了。

这很容易，但并不牢靠，因为以后可能有人傻乎乎地添加非纯方法。为防范这种情况，可将实例变量声明为 final 的：

```
    public class Card {
        private final int rank;
        private final int suit;

        ...
    }
```

你依然可以在构造函数中给这些变量赋值，但如果有人编写试图修改这些变量的方法，编译器将报错。

12.6 Card数组

可创建 String 对象数组，同样，也可创建 Card 对象数组。下面的语句创建了一个数组，其中包含 52 个 Card 对象：

```
Card[] cards = new Card[52];
```

图 12-2 是这个数组的状态图。

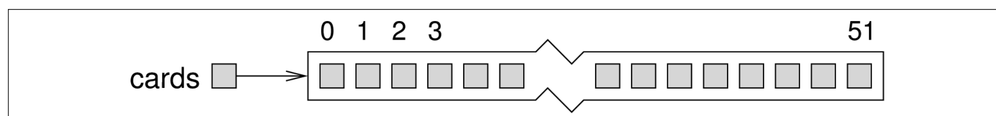


图 12-2：一个未填充的 Card 数组的状态图

虽然我们称之为“Card 对象数组”，但这个数组包含的是指向 Card 对象的引用，而不是 Card 对象本身。这个数组的元素被初始化为 null，可像通常那样访问：

```
if (cards[0] == null) {
    System.out.println("No card yet!");
}
```

然而，如果试图访问不存在的 Card 对象的实例变量，那么将引发 NullPointerException 异常。

```
cards[0].rank     // NullPointerException
```

用 Card 对象填充这个数组前，上述代码不可运行。编写嵌套 for 循环是填充数组的一种办法：

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
        cards[index] = new Card(rank, suit);
        index++;
    }
}
```

外面的循环迭代花色（从 0~3）。对于每种花色，内部的循环迭代点数（从 1~13）。由于外面的循环运行 4 次，里面的循环运行 13 次，因此循环体被执行 52 次。

我们使用了一个独立变量来跟踪下一张牌应存储在数组的什么地方，图 12-3 显示了创建开头两张牌后这个数组是什么样的。

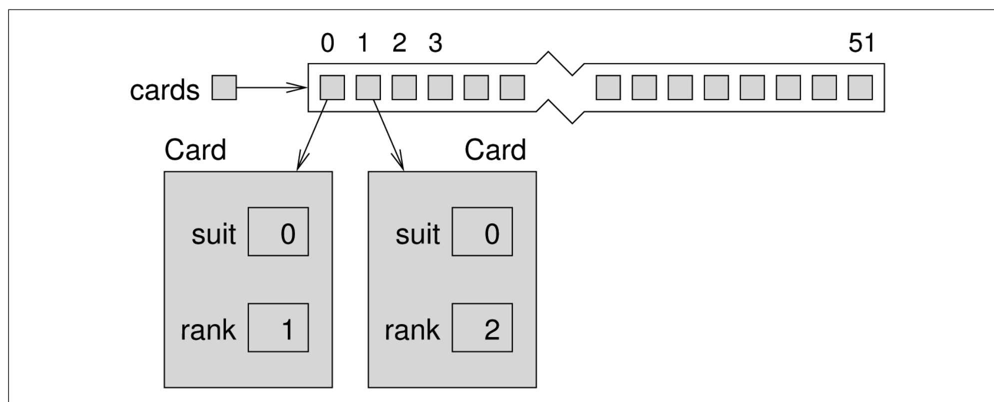


图 12-3：一个包含两张牌的 Card 数组的状态图

处理数组时，如果有一个显示其内容的方法将很方便。你已经见过很多次数组遍历模式，因此对下面的方法应该不会感到陌生：

```
public static void printDeck(Card[] cards) {
    for (int i = 0; i < cards.length; i++) {
        System.out.println(cards[i]);
    }
}
```

由于变量 cards 的类型为 Card[]，因此其元素的类型为 Card，所以 println 将调用 Card 类的方法 toString。上述方法的效果与调用 System.out.println(Arrays.toString(cards)) 类似。

12.7 顺序查找

接下来要编写的方法是 search，它接受两个参数——一个 Card 数组和一个 Card 对象，并返回 Card 对象在 Card 数组中的位置。如果 Card 对象没有出现在 Card 数组中，则返回−1。下面这个版本的 search 方法使用了 8.6 节中的算法——顺序查找（sequential search）：

```
public static int search(Card[] cards, Card target) {
    for (int i = 0; i < cards.length; i++) {
        if (cards[i].equals(target)) {
            return i;
        }
    }
    return -1;
}
```

这个方法发现指定的 Card 对象后立即返回，这意味着如果中途找到了目标，就不必遍历整个数组。如果未中途退出循环，就说明指定的 Card 对象不在数组中。注意，这个算法依赖于方法 equals。

如果数组中的 `Card` 对象未经排序，那么就无法以比顺序查找更快的速度进行查找，而必须检查每个 `Card` 对象，因为不这样做就无法确定要找的 `Card` 对象在不在数组中。然而，如果这些 `Card` 对象是按顺序排列的，就可使用更好的算法。

如何对数组排序将在下一章中介绍。将数组排序后，查找元素将容易得多。顺序查找的效率极低，数组非常大时尤其如此。

12.8 二分法查找

在字典中查找单词时，你不会从头到尾地逐页查找。由于单词是按字母顺序排列的，因此你很可能用*二分法查找*（binary search）算法。

(1) 从字典中间附近的某页开始。
(2) 将要查找的单词与该页的某个单词比较。如果找到，则就此结束。
(3) 如果这个单词排在要查找的单词前面，就往后翻并回到第 2 步。
(4) 如果该页的单词排在要查找的单词后面，就往前翻并回到第 2 步。

如果你在某页找到两个相邻的单词，即要查找的单词应排在它们之间，那么你就知道这本字典没有你要找的单词。

对于前面的 `Card` 对象数组，如果其中的 `Card` 对象是按顺序排列的，我们就可编写一个速度更快的 `search` 版本：

```
public static int binarySearch(Card[] cards, Card target) {
    int low = 0;
    int high = cards.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;                 // 第1步
        int comp = cards[mid].compareTo(target);

        if (comp == 0) {                            // 第2步
            return mid;
        } else if (comp < 0) {                      // 第3步
            low = mid + 1;
        } else {                                    // 第4步
            high = mid - 1;
        }
    }
    return -1;
}
```

我们首先声明了变量 `low` 和 `high`，用于表示要搜索的范围。一开始，我们搜索整个数组，从 `0` 到 `length - 1`。

在 `while` 循环中，我们重复二分法查找的 4 个步骤：

(1) 选择一个位于 `low` 和 `high` 之间的索引（`mid`），并将该索引处的 `Card` 对象同目标进行比较。

(2) 如果找到目标，就返回这个索引。
(3) 如果 `mid` 处的 `Card` 对象比目标小，就在范围 `mid + 1` 到 `high` 中搜索。
(4) 如果 `mid` 处的 `Card` 对象比目标大，就在范围 `low` 到 `mid - 1` 中搜索。

如果 `low` 大于 `high`，那么就意味着这个范围内没有任何 `Card` 对象，因此退出循环并返回 `-1`。注意，这个算法依赖于对象的方法 `compareTo`。

12.9 跟踪代码

为了搞明白二分法查找的工作原理，在循环开头添加如下打印语句大有帮助：

```
System.out.println(low + ", " + high);
```

如果像下面这样调用 `binarySearch`：

```
Card card = new Card(11, 0);
System.out.println(binarySearch(cards, card));
```

且要查找的 `Card` 对象位于索引 10 处，则输出如下：

```
0, 51
0, 24
0, 11
6, 11
9, 11
10
```

如果要查找的 Card 对象（如 `new Card(15, 1)`，即方块 15）不在数组中，输出将如下：

```
0, 51
26, 51
26, 37
26, 30
26, 27
-1
```

每执行一次循环，`low` 和 `high` 之间的距离都将减半。因此，经过 k 次迭代后，余下的 `Card` 对象数为 52/2k。要确定多少次迭代后查找才会结束，可求解方程 52/2k=1。结果为 log252，即大约 5.7。因此，可能只需要查看 5 或 6 个 `Card` 对象，而不需要像顺序查找那样查看全部的 52 个 `Card` 对象。

推而广之，如果数组包含 n 个元素，则二分法查找需要作 log2n 次比较，而顺序查找需要作 n 次比较。n 很大时，二分法查找的速度要快得多。

12.10 递归版本

也可以用递归方法实现二分法查找。为此可编写一个将 `low` 和 `high` 作为参数的方法，并将

第 3 步和第 4 步转换为递归调用。代码类似于下面这样：

```
public static int binarySearch(Card[] cards, Card target,
                               int low, int high) {
    if (high < low) {
        return -1;
    }
    int mid = (low + high) / 2;                              // 第1步
    int comp = cards[mid].compareTo(target);

    if (comp == 0) {                                         // 第2步
        return mid;
    } else if (comp < 0) {                                   // 第3步
        return binarySearch(cards, target, mid + 1, high);
    } else {                                                 // 第4步
        return binarySearch(cards, target, low, mid - 1);
    }
}
```

这里没有用 `while` 循环，而是用了一条 `if` 语句来终止递归。如果 `high` 小于 `low`，它们之间就不可能有 `Card` 对象，因此指定的 `Card` 对象肯定不在数组中。

编写递归程序时的两种常见错误是：忘记包含基线条件；编写的递归调用导致基线条件根本不可能满足。这两种错误都会导致无限递归，进而引发 `StackOverflowException` 异常。

12.11　术语表

- *编码*

  在两组值之间建立映射，以便用其中的一组值来表示另一组值。

- *类变量*

  类中被声明为 `static` 的变量；无论根据类创建了多少个对象，其类变量都只有一个副本。

- *顺序查找*

  查找数组元素的一种算法。它逐个查找，直到找到目标值为止。

- *二分法查找*

  查找有序数组的一种算法。它从中间元素开始，将该元素同目标进行比较，从而将余下的一半元素排除在外。

12.12　练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch12 中，有关如何下载这个仓库，请参阅

前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习12-1

12.6 节介绍了创建一副扑克牌的代码，请将这些代码封装到一个名为 `makeDeck` 的方法中，它不接受任何参数并返回一个填充好的 `Cards` 数组。

练习12-2

在有些扑克牌游戏中，A 指定的点数比 K 大。请修改方法 `compareTo` 以支持这种排序方式。

练习12-3

在扑克牌游戏中，同花指的是这样一手牌，即包含 5 张或更多同花色的牌。一手牌可包含任意数量的牌。

(1) 编写一个名为 `suitHist` 的方法，让它将一个扑克牌数组作为参数，并返回这手牌的花色直方图。你的解决方案只能遍历这个数组一次。

(2) 编写一个名为 `hasFlush` 的方法，让它将一个扑克牌数组作为参数，并在这手牌为同花时返回 `true`，否则返回 `false`。

练习12-4

如果能够在屏幕上显示扑克牌，那么将更有趣。如果你还没有阅读介绍 2D 图形的附录 B，应先阅读再来完成这个练习。本章示例代码所在的目录 ch12 包含：

- cardset-oxymoron，这个目录包含各种扑克牌图像；
- CardTable.java，一个演示如何读取并显示图像的示例程序。

CardTable.java 演示了如何使用二维数组，具体地说是二维图像数组。这个数组的声明类似于下面这样：

```
private Image[][] images;
```

变量 `images` 指向一个二维的 `Image` 对象数组，其中的 `Image` 对象是在 `java.awt` 包中定义的。创建这种数组的代码如下：

```
images = new Image[14][4];
```

这个数组包含 14 行（每种点数占 1 行，还有一个未用的、表示点数 0 的行）、4 列（每种花色一列）。下面的循环填充了这个数组：

```
String cardset = "cardset-oxymoron";
String suits = "cdhs";

for (int suit = 0; suit <= 3; suit++) {
    char c = suits.charAt(suit);
```

```
        for (int rank = 1; rank <= 13; rank++) {
            String s = String.format("%s/%02d%c.gif",
                                     cardset, rank, c);
            images[rank][suit] = new ImageIcon(s).getImage();
        }
    }
```

变量 cardset 包含扑克牌图像文件所在目录的名称。suits 是一个字符串，包含各种花色的单字母缩写。这两个变量被用来设置变量 s 的值，使其依次为各个扑克牌图像的文件名。例如，rank=1 且 suit=2 时，变量 s 的值为 "cardset-oxymoron/01h.gif"，这是红心 A 的图像。

循环中的最后一行代码读取图像文件并创建一个 Image 对象，再将其赋给索引为 rank 和 suit 的数组元素。例如，红心 A 的图像存储在索引 1 和索引 2 指定的位置。

如果编译并运行 CardTable.java，你将在一个绿色表格中看到整副扑克牌的图像。你可以用这个类来实现自己的扑克牌游戏。

第13章

数组对象

在前一章中，我们定义了一个表示单张扑克牌的类，并用一个 `Card` 对象数组来表示整副牌。

在本章中，我们将向面向对象编程再迈进一步，定义一个表示整副扑克牌的类。另外，我们还将介绍洗牌算法和数组排序算法。

本章的示例代码位于文件 Card.java 和 Deck.java 中，这些文件可在本书的代码仓库目录 `ch13` 中找到。有关如何下载这个代码仓库，请参阅前言中的“使用示例代码”一节。

13.1 Deck类

本章的主要目标是创建一个 `Deck` 类，用来封装一个 `Card` 对象数组。这个类的初始定义类似于下面这样：

```
public class Deck {
    private Card[] cards;

    public Deck(int n) {
        this.cards = new Card[n];
    }
}
```

其中的构造函数将实例变量初始化为一个包含 `n` 个 `Card` 对象的数组，但没有创建任何 `Card` 对象。图 13-1 显示了不包含任何 `Card` 对象的 `Deck` 对象是什么样的。

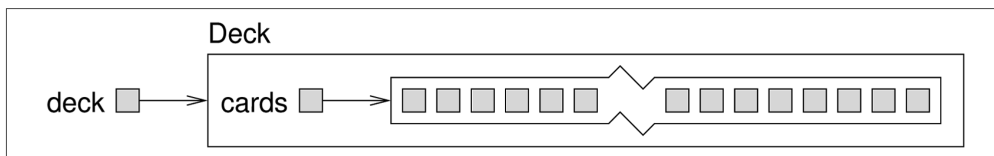


图 13-1：未填充的 Deck 对象的状态图

我们将添加另一个构造函数，让它创建一副标准牌（包含 52 张牌），并用 Card 对象填充它：

```
public Deck() {
    this.cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            this.cards[index] = new Card(rank, suit);
            index++;
        }
    }
}
```

这个方法类似于 12.6 节中的示例；我们只是将这个示例转换成了构造函数。现在可以像下面这样来创建一副标准的扑克牌：

```
Deck deck = new Deck();
```

创建 Deck 类后，就可将与整副扑克牌相关的方法放在这个类中了。回顾前面编写的方法，应放在这个类中的方法显然是 12.6 节中的 printDeck。

```
public void print() {
    for (int i = 0; i < this.cards.length; i++) {
        System.out.println(this.cards[i]);
    }
}
```

将静态方法转换为实例方法后，其代码通常会更短。要调用这个实例方法，只需要使用 deck.print() 即可。

13.2　洗牌

大多数扑克牌游戏需要提供洗牌功能，即将牌随机地排列。8.7 节介绍了如何生成随机数，但如何用随机数来洗牌并不那么简单。

一种选择是模拟人工洗牌的方式：通常将整副牌分成两半，再交替地从这两半中取牌。因为人工洗牌通常不能洗得很乱，所以需要重复大约 7 次才能让整副牌的排列顺序相当随机。

但计算机程序有个令人讨厌的特征，就是每次洗牌都能洗得很乱，但不是完全随机的。实际上，程序连续洗 8 次后，牌就会恢复到原来的顺序！有关这方面的更详细信息，请参阅 https://en.wikipedia.org/wiki/Faro_shuffle。

一种更佳的洗牌算法是遍历整副牌——每次一张，并在每次迭代中交换两张牌的位置。这种算法的工作原理大致如下。为了大致地描绘程序，我们结合使用了 Java 语句和自然语言，这种组合被称为*伪代码*（pseudocode）：

```
for each index i {
    // 在i和length - 1之间随机地选择一个数字
    // 将第i张牌和随机选择的那张牌的位置互换
}
```

伪代码的优点在于，让你能够清楚地知道需要哪些方法。从上述伪代码可知，你需要两个方法：一个方法要在 `low` 和 `high` 之间随机地选择一个数字，另一个方法要接受两个索引并交换这两个索引处的扑克牌。这样的方法被称为*辅助方法*（helper method），因为它们用于帮助实现更复杂的算法。

这种先编写伪代码，再编写所需方法的过程被称为*自上而下的开发*（top-down development），更详细的信息请参阅 https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design。

本章末尾有一个练习，要求你编写辅助方法 `randomInt` 和 `swapCards`，并用它们实现 `shuffle`。

13.3 选择排序

将整副牌洗乱后，你需要一个恢复排列顺序的方法。具有讽刺意义的是，有一种排序算法与洗牌算法类似。该排序算法被称为*选择排序*（selection sort），因为它反复遍历数组，每次都在余下的牌中选择最小（或最大）的那张。

在第一次迭代中，我们找到最小的牌，并将其与索引 0 处的牌互换位置；在第 i 次迭代中，我们在索引 i 右边的牌中找出最小的那张，并将其与索引 i 处的牌互换位置。下面是选择排序的伪代码：

```
public void selectionSort() {
    for each index i {
        // 在整个数组或索引i右边的范围内找到最小的那张牌
        // 将索引i处的牌与找到的最小的牌互换位置
    }
}
```

同样，这些伪代码可以帮助你设计辅助方法。在这个算法中，我们可以重用 `swapCards`，

因此只需要一个找到最小牌的方法——我们称之为 indexLowest。

本章末尾有一个练习，要求你编写辅助方法 indexLowest，并用它实现 selectionSort。

13.4 合并排序

选择排序算法很简单，但效率不高。要想对 n 个元素进行排序，它必须遍历数组 n-1 次。每次遍历所需要的时间与 n 成正比，因此总时间与 n^2 成正比。

在接下来的两节中，我们将实现一种效率更高的算法——合并排序（merge sort）。对 n 个元素进行排序时，合并排序所需的时间与 $n\log_2 n$ 成正比。这看似没什么大不了，但当 n 很大时，n^2 和 $n \log_2 n$ 的差别将非常大。

例如，$\log_2 1000000$ 的值大约为 20。因此，如果要对 100 万个数字进行排序，选择排序需要 1 万亿步，而合并排序只需 2000 万步。

合并排序的理念如下：如果有两堆已排好序的牌，将它们合并成排好序的一堆牌将既容易又快捷。请用一整副牌来尝试这种算法。

(1) 将整副牌分成两堆，每堆大约 26 张；分别对每堆牌进行排序，使其面朝上时最小的牌位于最上面；将两堆牌都面朝上放在你面前。
(2) 比较两堆牌最上面的那张，选择较小的，将其翻过来并加入合并堆。
(3) 重复第 2 步，直到其中一堆没牌，再将另一堆中余下的牌全都翻过来并加入合并堆。

结果是排好序的整副牌。接下来的几节将介绍如何用 Java 实现这种算法。

13.5 方法subdeck

合并算法的第一步是将整副牌分成两堆，每堆大约为半副牌。因此，我们可能需要一个名为 subdeck 的方法，让它接受两个指定索引范围的值，并返回新的 Deck 对象，其中包含指定范围内的牌：

```
public Deck subdeck(int low, int high) {
    Deck sub = new Deck(high - low + 1);
    for (int i = 0; i < sub.cards.length; i++) {
        sub.cards[i] = this.cards[low + i];
    }
    return sub;
}
```

第 1 行创建了一个未填充的 Deck 对象。在 for 循环中，复制当前 Deck 对象中的引用，并用它们填充新创建的 Deck 对象。

新 Deck 对象中的数组长度为 high - low + 1，因为索引为 low 和 high 的牌都包含在内。这种计算可能令人迷惑，如果忘记加 1，将导致“差一”错误。通常来说，绘图是避免这

种错误的最佳方式。

图 13-2 所示的状态图显示了 `low = 0` 和 `high = 4` 时将创建的新 `Deck` 对象。这个对象包含与原始 `Deck` 对象共享的 5 张牌，即它们互为别名。

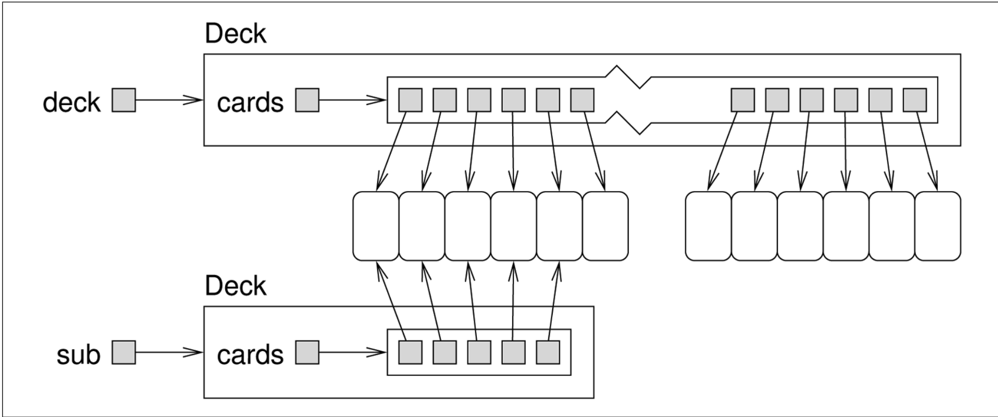


图 13-2：对 subdeck 带来的影响进行说明的状态图

互为别名可能不是什么好主意，因为修改共享的牌将影响多个 `Deck` 对象。然而，因为 `Card` 对象是不可修改的，所以就这里而言，互为别名不会带来任何问题。

13.6　方法merge

我们需要的下一个辅助方法是 `merge`，让它接受两个排好序的 `Deck` 对象，并返回一个新的 `Deck` 对象，新对象包含前两个 `Deck` 对象中的所有扑克牌，且这些扑克牌是按顺序排列的。这个方法的伪代码类似于下面这样（其中 `d1` 和 `d2` 是要进行合并排序的 `Deck` 对象）：

```
public static Deck merge(Deck d1, Deck d2) {
    // 新建一个可容纳所有扑克牌的Deck对象

    // 用索引i和j来分别记录当前我们位于两个传入的Deck对象的什么位置
    int i = 0;
    int j = 0;

    // 用索引k来遍历新创建的Deck对象(result)
    for (int k = 0; k < result.cards.length; k++) {

        // 如果d1为空,则d2获胜
        // 如果d2为空,则d1获胜
        // 如果d1和d2都不为空,则对两张牌进行比较

        // 将获胜的牌加入新创建的Deck对象中,并放在索引k指定的位置
        // 将i或j加1
    }
```

```
        // 返回新创建的Deck对象
    }
```

本章末尾有一个练习，要求你实现方法 merge。

13.7　添加递归

确定方法 merge 能够正确地运行后，就可尝试开发合并排序算法的版本了：

```
    public Deck almostMergeSort() {
        // 将Deck对象一分为二
        // 用selectionSort对这两部分分别排序
        // 合并两部分并返回结果
    }
```

本章末尾有个练习，要求你实现这种算法。正确实现该算法后，有趣的部分就开始了！合并排序的神奇之处在于，它天然就是递归的。

将各部分排序时，为何要调用较慢的算法 selectionSort 呢？为何不调用你正在编写的 mergeSort 呢？这不仅是个不错的主意，要想获得 $\log_2$ 性能优势，你也必须这样做。

要让 mergeSort 递归地运行，必须添加基线条件，否则将没完没了地递归。一个简单的基线条件是有一部分没有牌或只包含 1 张牌。mergeSort 收到这样小的部分时，可原封不动地将其返回，因为它已经是排好序的。

递归版 mergeSort 应类似于下面这样：

```
    public Deck mergeSort() {
        // 如果Deck对象中没有牌或只包含一张牌,就原封不动地返回它
        // 将Deck对象分成两部分
        // 用mergeSort分别对这两部分排序
        // 合并这两部分并返回结果
    }
```

与通常情况一样，你也可以用两种方式研究递归程序：研究整个执行流程或采取“姑且相信”的态度（参见 6.8 节）。就这个示例而言，你完全可以采取姑且相信的态度。

用 selectionSort 对各个部分进行排序时，不必研究整个执行流程。可以假定 selectionSort 能够正确地运行，因为你之前调试过了。为将 mergeSort 变成递归的，你只是将一种排序算法替换成了另一种排序算法，因此没有理由采用不同的方式来研究程序。

准确地说，差不多是这样的，因为你可能需要确保基线条件没问题且它终将得以满足。除此之外，递归版本与非递归版本没有其他不同。

13.8 术语表

- 伪代码

  一种程序设计方式，结合自然语言和 Java 语言来设计大致的草案。

- 辅助方法

  通常是较小的方法，本身所做的工作微不足道，只是为更复杂的方法提供帮助。

- 自上而下的开发

  将问题分成几个小问题，再每次解决一个。

- 选择排序

  一种简单的排序算法，执行 *n* 次最小或最大元素查找。

- 合并排序

  一种递归的排序算法，将数组分成两部分，用合并排序分别对每部分进行排序，再合并结果。

13.9 练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch13 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习13-1

要想更详细地了解本章介绍的排序算法以及其他排序算法，可参阅 http://www.sorting-algorithms.com/。这个网站对这些算法作了诠释，包含演示工作原理的动画，还对这些算法的效率作了分析。

练习13-2

这个练习旨在实现本章介绍的洗牌算法。

(1) 在本书的代码仓库中，有一个名为 Deck.java 的文件，其中包含了本章的示例代码。确认这个文件在你使用的开发环境中能够通过编译。

(2) 在 `Deck` 类中添加一个名为 `randomInt` 的方法，让它接受两个 `int` 参数（`low` 和 `high`），并返回一个位于 `low` 和 `high` 之间（闭区间）的随机整数。可用 `java.util.Random` 提供的 `nextInt`，这个方法在 8.7 节中介绍过。但你应避免每次调用 `randomInt` 时都创建一个 `Random` 对象。

(3) 编写一个名为 `swapCards` 的方法，让它接受两个索引，并互换这两个索引处扑克牌的位置。

(4) 编写一个名为 `shuffle` 的方法，在其中实现 13.2 节介绍的算法。

练习13-3

这个练习旨在实现本章介绍的排序算法。可用前一个练习中的文件 Deck.java，也可重新创建一个。

(1) 编写一个名为 `indexLowest` 的方法，让它用方法 `compareCard` 找出给定范围（从 `lowIndex` 到 `highIndex` 的闭区间）内的最小牌。
(2) 编写一个名为 `selectionSort` 的方法，在其中实现 13.3 节介绍的选择排序算法。
(3) 根据 13.4 节的伪代码编写一个名为 `merge` 的方法。为对你编写的方法进行测试，最佳的方式是创建一个 `Deck` 对象，并对其执行洗牌操作，再用 `subdeck` 将它分成两部分，并用 `selectionSort` 分别对这两部分进行排序。然后就可将这两部分传递给方法 `merge`，看看它是否管用。
(4) 编写简单版 `mergeSort`，即将 `Deck` 对象分成两部分，用 `selectionSort` 对这两部分进行排序，再用 `merge` 创建一个排好序的新 `Deck` 对象。
(5) 编写递归版 `mergeSort`。别忘了，`selectionSort` 是一个非纯方法，而 `mergeSort` 是一个纯方法，这意味着调用它们的方式不同：

```
deck.selectionSort();      // 修改当前Deck对象
deck = deck.mergeSort();   // 将当前Deck对象替换为新的Deck对象
```

练习13-4

这个练习旨在通过实现插入排序来践行自上而下的编程。请访问 http://www.sorting-algorithms.com/insertion-sort，阅读其中对插入排序的介绍，再编写一个名为 `insertionSort` 的方法来实现这种算法。

练习13-5

为 `Deck` 类编写方法 `toString`。它应该返回一个对 `Deck` 对象中的扑克牌作了描述的字符串。该字符串的内容应与 13.1 节中的方法 `print` 的输出相同。

提示：可用运算符 + 来拼接字符串，但效率不太高。为提高效率，可考虑使用 `java.util.StringBuilder`；要查找有关这个类的文档，可在网上搜索 Java StringBuilder。

第14章

包含其他对象的对象

有了表示单张扑克牌和整副牌的类后，我们用它们来开发一款游戏！ *Crazy Eights* 是一款经典的扑克牌游戏，可供两个或更多人一起玩。玩家的主要目标是最先把手里的牌出完。这个扑克牌的玩法如下。

- 给每个人发 5 张或更多的牌，再发一张牌并将其翻开，作为弃牌堆的第一张牌。将余下的牌面朝下，作为储备牌。
- 每个人轮流出一张牌到弃牌堆。出的牌必须与前一张牌的点数或花色相同，或者为万能牌 8。
- 如果没有匹配的牌或 8 可出，玩家就必须从储备牌中取牌，直到取到匹配的牌或 8。
- 储备牌取光后，就对弃牌堆进行洗牌（最上面的那张除外），并将其用作储备牌。
- 一旦有人出完了手中的牌，游戏便结束，并对其他人罚分。罚多少分是根据手上余下的牌计算的：8 为 20 分，花牌 10 分，其他牌与其点数相同。

要想获悉更详细的细节，可参阅 https://en.wikipedia.org/wiki/Crazy_Eights；但前面的介绍已足够详细，可以开始编写这款游戏了。

本章的示例代码位于本书代码仓库的目录 ch14 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。

14.1　Deck和手里的牌

要实现这款游戏，需要表示整副牌、弃牌堆、储备牌以及每个人手中的牌；还需要能够发

牌、取牌和出牌。

前一章的 Deck 类可满足上述部分需求，但存在两个问题。

- 一手牌和一堆牌的规模不同，且它们的规模会随游戏的进行而不断变化。前一章的 Deck 类的实现使用了一个 Card 数组，而数组的长度是固定的。
- 不知道用 Deck 对象表示一手牌和一堆牌是否合适。我们可能需要创建新类来表示其他的扑克牌集合。

为解决第一个问题，可用 java.util 包中的 ArrayList 替换 Card 数组。ArrayList 是一种集合（collection），即包含其他对象的对象。

Java 类库提供了各种集合，就这里的目标而言，ArrayList 是不错的选择，因为它能自动增大和缩小，还提供了添加和删除元素的方法。

为解决第二个问题，可用一种名为*继承*（inheritance）的语言功能。我们将定义一个新类——CardCollection，用于表示扑克牌集合，再将 Deck 和 Hand 定义为 CardCollection 的子类。

子类（subclass）是“扩展”已有类的新类，即它拥有已有类的所有属性和方法，还新增了属性和方法。稍后将介绍详情，我们先来定义 CardCollection 类。

14.2 CardCollection

下面是 CardCollection 类的初始定义，它用的是 ArrayList 而不是基本数组类型：

```
public class CardCollection {

    private String label;
    private ArrayList<Card> cards;

    public CardCollection(String label) {
        this.label = label;
        this.cards = new ArrayList<Card>();
    }
}
```

声明 ArrayList 时，需要在尖括号（<>）中指定它将包含的对象的类型。上述声明指出 cards 是一个包含 Card 对象的 ArrayList。

构造函数接受一个字符串实参，并将其赋给实例变量 label；它还将 cards 初始化为一个空的 ArrayList。

ArrayList 提供了方法 add，用于在集合中添加元素。我们给 CardCollection 添加一个执行这种任务的方法：

```
public void addCard(Card card) {
    this.cards.add(card);
}
```

在本书前面，为方便识别属性，我们显式地使用了 this。在 addCard 和其他实例方法中，不使用关键字 this 也可访问实例变量，因此从现在开始，我们将省略 this：

```
public void addCard(Card card) {
    cards.add(card);
}
```

我们还需要提供将扑克牌从集合中删除的功能。下面的方法接受一个索引，删除指定位置的扑克牌，并将后面的扑克牌前移，以填补留下的空缺：

```
public Card popCard(int i) {
    return cards.remove(i);
}
```

从一副洗好的牌中发牌时，发哪张牌无所谓，但效率最高的做法是先发最后的牌，这样就不用移动余下的牌了。下面是被重载的方法 popCard 的版本之一，它删除并返回最后一张牌：

```
public Card popCard() {
    int i = size() - 1;
    return popCard(i);
}
```

注意，popCard 调用了 CardCollection 的方法 size，这个方法转而调用了 ArrayList 的方法 size：

```
public int size() {
    return cards.size();
}
```

出于方便考虑，CardCollection 还提供了方法 empty，它在 size 为 0 时返回 true：

```
public boolean empty() {
    return cards.size() == 0;
}
```

像 addCard、popCard 和 size 这样只是调用另一个方法，而本身所做不多的方法被称为包装器方法（wrapper method）。我们将用这些包装器方法来实现一些更重要的方法，如 deal：

```
public void deal(CardCollection that, int n) {
    for (int i = 0; i < n; i++) {
        Card card = popCard();
        that.addCard(card);
    }
}
```

方法 deal 将扑克牌从当前集合（this）中删除，并将其加入到通过参数（that）指定的集合中。第二个形参 n 指定要发多少张牌。

访问 ArrayList 的元素时，不能使用数组运算符 []，而必须使用方法 get 和 set。下面是一个调用 get 的包装器方法：

```
public Card getCard(int i) {
    return cards.get(i);
}
```

方法 last 获取最后一张牌，但不删除：

```
public Card last() {
    int i = size() - 1;
    return cards.get(i);
}
```

为对修改扑克牌集合的方式进行控制，我们没有提供调用 set 的包装器方法。在我们提供的方法中，只有两个版本的 popCard 以及下述版本的 swapCards 是非纯方法：

```
public void swapCards(int i, int j) {
    Card temp = cards.get(i);
    cards.set(i, cards.get(j));
    cards.set(j, temp);
}
```

我们用 swapCards 实现 shuffle，这在 13.2 节中讨论过：

```
public void shuffle() {
    Random random = new Random();
    for (int i = size() - 1; i > 0; i--) {
        int j = random.nextInt(i);
        swapCards(i, j);
    }
}
```

ArrayList 还提供了这里没有使用的其他方法。要了解这些方法，可查阅相关的文档；而要查找这些文档，可在网上搜索 Java ArrayList。

14.3　继承

至此，我们定义了一个表示扑克牌集合的类，下面用它定义 Deck 和 Hand 类。Deck 类的完整定义如下：

```
public class Deck extends CardCollection {

    public Deck(String label) {
        super(label);
```

```
        for (int suit = 0; suit <= 3; suit++) {
            for (int rank = 1; rank <= 13; rank++) {
                cards.add(new Card(rank, suit));
            }
        }
    }
}
```

第 1 行用关键词 extends 指出 Deck 类扩展了 CardCollection 类，这意味着 Deck 对象将包含 CardCollection 对象的所有实例变量和方法。换而言之，Deck“继承”了 CardCollection。我们还可以说 CardCollection 是一个*超类*（superclass），而 Deck 是其子类。

在 Java 中，类只能扩展一个超类。没有用 extends 指定超类的类将自动继承 java.lang.Object。因此，在这个实例中，Deck 扩展了 CardCollection，而 CardCollection 扩展了 Object。Object 类提供了默认方法 equals 和 toString 等。

构造函数不被继承，但所有其他的公有属性和方法都被继承。到目前为止，Deck 中新增的唯一一个方法是构造函数，因此你可像下面这样创建 Deck 对象：

```
Deck deck = new Deck("Deck");
```

Deck 构造函数的第 1 行使用了以前没有介绍过的 super，这是一个关键词，指的是当前类的超类。像这里这样将 super 用作方法时，将调用超类的构造函数。

因此在这里，super 调用 CardCollection 的构造函数，而这个构造函数初始化属性 label 和 cards。这个构造函数返回后，将接着执行 Deck 的构造函数：用 Card 对象填充原本为空的 ArrayList。

这就是 Deck 类。我们还需要表示一手牌和一堆牌，其中前者是玩家手里的扑克牌集合，而后者是桌子上的扑克牌集合。为此，我们可定义两个类，分别用于表示一手牌和一堆牌，但它们的差别不大。因此，我们将用同一个类 Hand 来表示一手牌和一堆牌。这个类的定义类似于下面这样：

```
public class Hand extends CardCollection {

    public Hand(String label) {
        super(label);
    }

    public void display() {
        System.out.println(getLabel() + ": ");
        for (int i = 0; i < size(); i++) {
            System.out.println(getCard(i));
        }
        System.out.println();
```

```
    }
}
```

与 Deck 一样，Hand 也扩展了 CardCollection，因此它继承了 getLabel、size 和 getCard 等方法，并在方法 display 中使用了这些方法。Hand 也提供了一个构造函数，就只是调用 CardCollection 的构造函数，除此之外什么都没有做。

总之，Deck 类似于 CardCollection，但提供的构造函数不同。且 Hand 也类似于 CardCollection，但提供了一个额外的方法——display。

14.4 发牌

现在可以创建一个 Deck 对象并发牌了。下面是一个简单示例，它将 5 张牌作为玩家手中的牌，并将余下的牌作为储备牌：

```
Deck deck = new Deck("Deck");
deck.shuffle();

Hand hand = new Hand("Hand");
deck.deal(hand, 5);
hand.display();

Hand drawPile = new Hand("Draw Pile");
deck.dealAll(drawPile);
System.out.printf("Draw Pile has %d cards.\n",
                  drawPile.size());
```

CardCollection 提供了方法 dealAll，这个方法将余下的牌都发出去。这个示例的输出如下：

```
Hand:
5 of Diamonds
Ace of Hearts
6 of Clubs
6 of Diamonds
2 of Clubs

Draw Pile has 47 cards.
```

当然，如果你运行这个示例，得到的那手牌可能不同，因此整副牌洗得很乱。

如果你很细心，可能注意到了一些怪异的情况。我们再来看一眼方法 deal 的定义：

```
public void deal(CardCollection that, int n) {
    for (int i = 0; i < n; i++) {
        Card card = popCard();
        that.addCard(card);
    }
}
```

注意，第一个形参是一个 CardCollection 对象，而我们却以如下方式调用这个方法：

```
Hand hand = new Hand("Hand");
deck.deal(hand, 5);
```

相应的实参是一个 Hand 对象，而不是 CardCollection 对象。这种做法怎么就合法呢？这是因为 Hand 是 CardCollection 的子类，所以 Hand 对象也被视为 CardCollection 对象。如果方法需要一个 CardCollection 对象，可向它提供一个 Hand、Deck 或 CardCollection 对象。

然而，反过来则不行：并非每个 CardCollection 对象都是 Hand 对象。因此，如果方法需要一个 Hand 对象，你就必须给它提供一个 Hand 对象，而不能给它提供一个 CardCollection 对象。

一个对象可以属于多种类型好像有点怪，但现实世界也存在这样的情况。所有的猫都是哺乳动物，所有哺乳动物都是动物；但并非所有的动物都是哺乳动物，并非所有的哺乳动物都是猫。

14.5 Player类

前面定义的类可用于任何扑克牌游戏，即我们没有在这些类中实现任何 *Crazy Eights* 特有的规则。这可能是件好事，因为以后要开发另一款游戏时，可轻松地重用这些类。

接下来该实现这些规则了。为此，我们将使用两个类：Player 和 Eights，前者封装了玩家策略，后者创建并维护游戏的状态。下面列出了 Player 类定义的开头部分：

```
public class Player {

    private String name;
    private Hand hand;

    public Player(String name) {
        this.name = name;
        this.hand = new Hand(name);
}
```

Player 有两个私有属性：name 和 hand。构造函数接受玩家的名字，并将其存储到实例变量 name 中。在这里，我们必须用 this 将同名的实例变量和形参区分开来。

Player 提供的主要方法是 play；每当轮到玩家出牌时，都用它来决定出哪张牌：

```
public Card play(Eights eights, Card prev) {
    Card card = searchForMatch(prev);
    if (card == null) {
        card = drawForMatch(eights, prev);
    }
```

```
        return card;
    }
```

第一个形参是一个引用，指向封装了游戏状态的 Eights 对象。需要取牌时要用到这个对象。第二个形参 prev 指的是弃牌堆最上面的扑克牌。

我们采用自上而下的开发，让 play 调用两个辅助方法——searchForMatch 和 drawForMatch。searchForMatch 在当前玩家手中查找与前一个玩家所出的牌匹配的牌：

```
    public Card searchForMatch(Card prev) {
        for (int i = 0; i < hand.size(); i++) {
            Card card = hand.getCard(i);
            if (cardMatches(card, prev)) {
                return hand.popCard(i);
            }
        }
        return null;
    }
```

这里的策略非常简单：for 循环找到并返回第一张匹配的。如果没有匹配的牌，就返回 null。在这种情况下，就必须不断取牌，直到获得匹配的牌：

```
    public Card drawForMatch(Eights eights, Card prev) {
        while (true) {
            Card card = eights.draw();
            System.out.println(name + " draws " + card);
            if (cardMatches(card, prev)) {
                return card;
            }
            hand.addCard(card);
        }
    }
```

其中的 while 循环将不断运行，直到取到匹配的牌（这里暂时假设终将取到匹配的牌）。它用 Eights 对象来取一张牌。如果这张牌是匹配的，就返回该牌；否则就将其放到玩家手里，并继续取牌。

searchForMatch 和 drawForMatch 都使用了 cardMatches。cardMatches 是一个静态方法，也是在 Player 类中定义的，其以简单的方式实现了这个游戏的规则：

```
    public static boolean cardMatches(Card card1, Card card2) {
        if (card1.getSuit() == card2.getSuit()) {
            return true;
        }
        if (card1.getRank() == card2.getRank()) {
            return true;
        }
        if (card1.getRank() == 8) {
            return true;
        }
```

```
        return false;
    }
```

最后，Player 还提供了 score，它在游戏结束时根据玩家手里余下的牌计算罚分：

```
    public int score() {
        int sum = 0;
        for (int i = 0; i < hand.size(); i++) {
            Card card = hand.getCard(i);
            int rank = card.getRank();
            if (rank == 8) {
                sum -= 20;
            } else if (rank > 10) {
                sum -= 10;
            } else {
                sum -= rank;
            }
        }
        return sum;
    }
```

14.6 Eights类

13.2 节介绍了自上而下的开发，这种程序开发方式确定高层次的目标，如洗牌，并将其分解为更小的问题，如在数组中查找最小的元素或交换两个元素。

本节将介绍自下而上的开发（bottom-up development），它反过来做，即先找出需要的简单部分，再将它们组装成更复杂的算法。

根据 *Crazy Eights* 的规则，可确定需要的一些方法。

- 创建整副扑克牌、弃牌堆、储备牌以及表示玩家的对象。
- 发牌。
- 检查游戏是否结束。
- 如果没有了储备牌，就将弃牌堆洗一下，并将其作为储备牌。
- 取牌。
- 跟踪轮到谁出牌以及从一个玩家切换到下一个玩家。
- 显示游戏的状态。
- 进入下一轮前等待用户。

现在可以开始实现这些部分了。下面是封装游戏状态的 Eights 类定义的开头部分：

```
    public class Eights {

        private Player one;
        private Player two;
        private Hand drawPile;
```

```
    private Hand discardPile;
    private Scanner in;
```

这个版本只有两个玩家。本章末尾有一个练习，要求你修改这些代码，以支持更多玩家。

最后一个实例变量是一个 Scanner 对象，我们将在每次出牌后用它提示用户输入。下面的构造函数初始化所有的实例变量并发牌：

```
public Eights() {
    Deck deck = new Deck("Deck");
    deck.shuffle();

    int handSize = 5;
    one = new Player("Allen");
    deck.deal(one.getHand(), handSize);
    two = new Player("Chris");
    deck.deal(two.getHand(), handSize);

    discardPile = new Hand("Discards");
    deck.deal(discardPile, 1);

    drawPile = new Hand("Draw pile");
    deck.dealAll(drawPile);

    in = new Scanner(System.in);
}
```

接下来需要实现的是检查游戏是否结束的方法。只要有一个玩家手里没牌，游戏便结束了：

```
public boolean isDone() {
    return one.getHand().empty() || two.getHand().empty();
}
```

没有储备牌时，我们必须将弃牌堆洗一下。完成这种任务的方法如下：

```
public void reshuffle() {
    Card prev = discardPile.popCard();
    discardPile.dealAll(drawPile);
    discardPile.addCard(prev);
    drawPile.shuffle();
}
```

第 1 行保存 discardPile 最上面的那张牌；第 2 行将余下的牌转移到 drawPile。接下来，我们将保存的牌放回到 discardPile，并对 drawPile 执行洗牌操作。

现在可以在 draw 中调用 reshuffle 了：

```
public Card draw() {
    if (drawPile.empty()) {
        reshuffle();
```

```
        }
        return drawPile.popCard();
    }
```

要从一个玩家切换到另一个玩家，我们可以像下面这样做：

```
    public Player nextPlayer(Player current) {
        if (current == one) {
            return two;
        } else {
            return one;
        }
    }
```

方法 nextPlayer 将当前玩家作为参数，并返回接下来轮到的玩家。

最后两个部分是 displayState 和 waitForUser：

```
    public void displayState() {
        one.display();
        two.display();
        discardPile.display();
        System.out.println("Draw pile:");
        System.out.println(drawPile.size() + " cards");
    }

    public void waitForUser() {
        in.nextLine();
    }
```

可用这些方法编写 takeTurn，用来执行玩家的一次出牌过程：

```
    public void takeTurn(Player player) {
        Card prev = discardPile.last();
        Card next = player.play(this, prev);
        discardPile.addCard(next);

        System.out.println(player.getName() + " plays " + next);
        System.out.println();
    }
```

takeTurn 读取弃牌堆最上面的那张牌，将其传递给前一节介绍过的 player.play，并将它返回的牌加入弃牌堆。

最后，我们用 takeTurn 和其他方法来编写 playGame：

```
    public void playGame() {
        Player player = one;

        // 不断地玩,直到有人获胜
        while (!isDone()) {
            displayState();
```

```
            waitForUser();
            takeTurn(player);
            player = nextPlayer(player);
        }

        // 显示最终得分
        one.displayScore();
        two.displayScore();
    }
```

大功告成！注意，自下而上开发的结果与自上而下类似：有一个调用辅助方法的高级方法。主要差别在于实现解决方案的顺序不同。

14.7　类之间的关系

本章演示了两种常见的类间关系。

- 组合

  一个类的实例包含另一个类的实例的引用。例如，`Eights` 实例包含两个指向 `Player` 对象的引用、两个指向 `Hand` 对象的引用以及一个指向 `Scanner` 对象的引用。

- 继承

  一个类扩展了另一个类。例如，`Hand` 扩展了 `CardCollection`，因此每个 `Hand` 实例都是 `CardCollection` 对象。

组合关系也被称为有一个（HAS-A）关系，如 `Eights` 有一个 `Scanner`；继承关系也被称为是一个（IS-A）关系，如 `Hand` 是一个 `CardCollection`。这些术语让你能够以简洁的方式讨论面向对象设计。

对于这些关系，还有一种标准的图形表示方式，那就是 UML 类图。正如你在 10.9 节看到的，类的 UML 表示是一个方框，这个方框包含三部分：类名、属性和方法。但用于显示类之间的关系时，后两部分是可选的。

类之间的关系用箭头表示：标准箭头表示组合关系，空三角形箭头（通常是向上的）表示继承关系。图 14-1 显示了本章定义的类以及它们之间的关系。

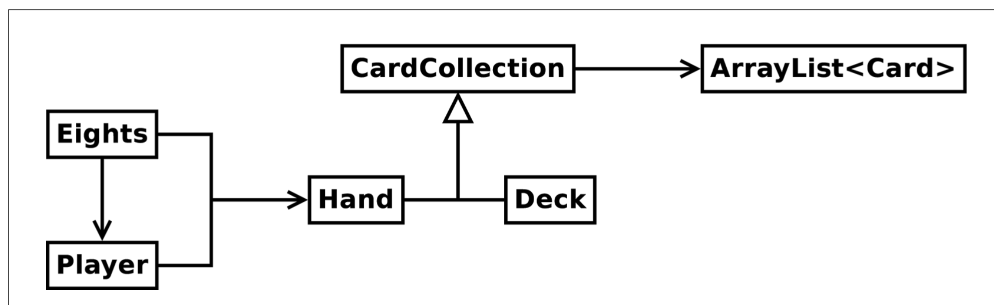


图 14-1：显示本章定义的类及其关系的 UML 图

UML 属于国际标准，因此几乎任何软件工程师看到这种图时都能够明白它所表示的设计。类图只是 UML 标准定义的众多图形表示之一。

本章总结了本书介绍的各种技术，包括变量、方法、条件、循环、数组、对象和算法，但愿对你有所帮助。祝贺你阅读完了本书！

14.8　术语表

- 集合

  包含其他对象的对象，更具体地说是 Java 库中包含对象的对象，如 `ArrayList`。

- 继承

  一种定义新类的方式，让新类包含既有类的所有实例变量和方法。

- 子类

  继承或扩展既有类的类。

- 超类

  被另一个类扩展的既有类。

- 包装器方法

  调用另一个方法，但除此之外所做工作不多的方法。

- 自下而上的开发

  一种程序开发方式，先确定并实现简单的部分，再将它们组合成更复杂的算法。

- 有一个

  两个类之间的关系，其中一个类将另一个类的实例作为属性。

- 是一个

  两个类之间的关系，其中一个类扩展了另一个类，即子类的实例也是超类的实例。

14.9　练习

本章的示例代码位于仓库 ThinkJavaCode 的目录 ch14 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。做以下的练习前，建议你先编译并运行本章的示例。

练习14-1

在方法 `Player.play` 中实现一种更佳的策略。例如，如果有多张牌匹配，且其中一张为 8，那么就出 8。

想想其他能最大限度地减少罚分的策略，如优先考虑出点数最大的牌。编写一个扩展 `Player` 的新类，并重写方法 `play`，以实现你制定的出牌策略。

练习14-2

编写一个循环，在其中玩 100 次本章介绍的游戏，并记录每位玩家分别赢了多少次。如果你实现了前一个练习介绍的多种出牌策略，那么可让这些策略进行较量，看看哪种策略的效果最好。

提示：设计一个扩展 `Player` 的 `Genius` 类，并在其中重写方法 `play`，再将一个玩家声明为 `Genius` 对象。

练习14-3

本章编写的程序存在一个缺陷，那就是只支持两个玩家。请修改 `Eights` 类，在其中定义一个包含玩家的 `ArrayList`，并相应地修改选择下一个玩家的方法 `nextPlayer`。

练习14-4

设计本章的程序时，我们力图将类数量降到最少，这导致有些方法不太合理。例如，`cardMatches` 是 `Player` 的一个静态方法，但将其作为 `Card` 的实例方法更合适。

问题是我们想让 `Card` 类可用于任何扑克牌游戏，而不仅仅是 `Crazy Eights` 中。为解决这种矛盾，可添加一个扩展 `Card` 的新类——`EightsCard`，并在其中定义方法 `match`，让它根据 `Crazy Eights` 的规则来检查两张牌是否匹配。

另外，你还可创建一个扩展 `Hand` 的新类——`EightsHand`，并在其中定义方法 `scoreHand`，让它累计手中所有牌的罚分。同时，顺便在 `EightsCard` 中添加一个名为 `scoreCard` 的方法。

无论你是否做了这些修改，都请绘制一个 UML 类图来显示这种继承层次结构。

附录A

开发工具

编译、运行和调试 Java 代码的步骤随使用的开发环境和操作系统而异。我们没有将这些细节放在正文中，因为这会分散读者的注意力。相反，我们专辟了这个附录，简要地介绍 DrJava——一个非常适合初学者使用的集成开发环境（integrated development environment，IDE），以及用于检查代码质量的 Checkstyle 和用于测试的 JUnit 等工具。

A.1　安装DrJava

要想用 Java 进行编程，最简单的方式是使用在浏览器中编译和运行 Java 代码的网站，如 jdoodle.com、compilejava.net 和 tutorialspoint.com 等。

如果你不能在计算机上安装软件（公立学校和网吧就属于这种情况），可用这些在线开发环境来完成本书的大部分示例。

然而，如果你要在自己的计算机上编译并运行 Java 程序，就需要：

- Java 开发包（Java Development Kit，JDK），它自带了编译器、对编译得到的字节码进行解释的 Java 虚拟机（Java Virtual Machine，JVM）以及 Javadoc 等其他工具；
- 简单的文本编辑器（text editor，如 Notepad++ 或 Sublime Text）和 / 或 IDE（如 DrJava、Eclipse、jGrasp 或 NetBeans）。

对于 JDK，我们推荐使用 Oracle 免费提供的 Java SE 标准版；对于 IDE，我们推荐使用 DrJava，这是一个用 Java 编写的开源开发环境，如图 A-1 所示。

要想安装 JDK，可在网上搜索 download JDK，这将把你带到 Oracle 网站。向下滚动到 Java Platform, Standard Edition 部分，并单击 JDK 下方的 download 按钮。再选择单选按钮 Accept License Agreement，并单击用于你的操作系统的安装程序。下载安装程序后，别忘了安装！

要想安装 DrJava，请访问 http://drjava.org，并下载 JAR 文件。建议你将这个文件保存到桌面或其他方便的地方；下载完毕后，单击 JAR 文件来运行 DrJava。更多详细信息请参阅 DrJava 文档（http://drjava.org/docs/quickstart/）。

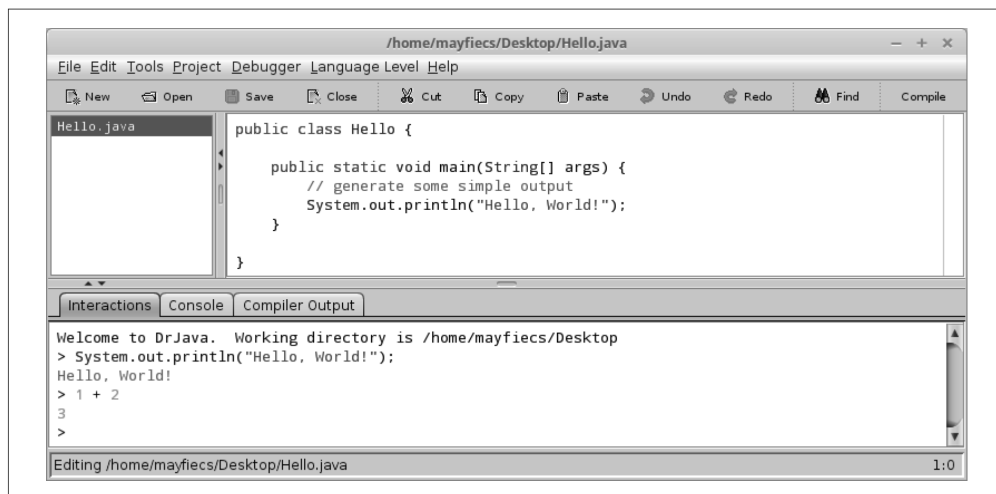


图 A-1：在 DrJava 中编辑程序 Hello World

首次运行 DrJava 时，建议你选择菜单 Edit>Preferences，并修改 Miscellaneous 下的三项设置：将 Indent Level（Tab 键对应的缩进量）设置为 4；选择复选框 Automatically Close Block Comments（自动添加块注释结束标志）；取消选择复选框 Keep Emacs-style Backup Files（保存 Emacs 风格的备份文件）。

A.2　DrJava Interactions窗格

DrJava 最有用的功能之一是窗口底部的 Interactions 窗格，它让你无需编写类定义再保存、编译并运行程序就能尝试运行代码，如图 A-2 所示。

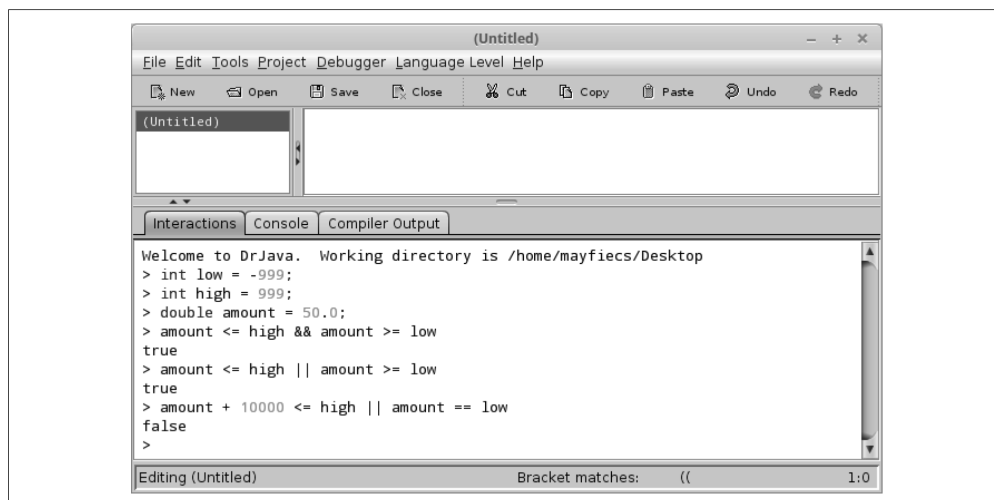


图 A-2：DrJava 中的 Interactions 窗格

使用 Interactions 窗格时，需要注意的一个细节是，如果表达式（或语句）末尾没有分号，DrJava 将自动显示其值。注意，在图 A-2 中，开头的变量声明以分号结尾，但接下来几行中的逻辑表达式末尾没有分号。这让你不用每次都输入 `System.out.println`。

使用 `Interactions` 窗格的优点在于，无需创建新类、声明 `main` 方法、在 `System.out.println` 语句中添加表达式、保存源代码文件再编译代码。另外，还可通过按电脑键盘上的向上 / 下箭头键来重复以前的命令以及逐渐修改代码。

A.3　命令行界面

你可以学习的一项最强大、最有用的技能是如何使用命令行界面（command-line interface），也叫终端。命令行是直通操作系统的接口，让你能够运行程序、管理文件和目录以及监视系统资源。很多用于软件开发和通用计算的高级工具都只能通过命令行界面来使用。

网上有很多有关如何使用命令行的优秀教程，在网上搜索 command line tutorial 就可找到。在 Linux 和 OS X 等 Unix 系统中，只需掌握四个命令就可开始使用命令行界面：切换工作目录的命令（`cd`）、列出目录内容的命令（`ls`）、编译 Java 程序的命令（`javac`）以及运行 Java 程序的命令（`java`）。

图 A-3 演示了如何执行这些命令，其中源代码文件 Hello.java 存储在目录 Desktop 中。切换到这个目录并列出其中的文件后，我们用命令 `javac` 来编译 Hello.java；然后，再次执行命令 `ls`，并发现编译器生成了一个新文件——Hello.class，它包含字节码。我们用命令 `java` 运行这个程序，其输出显示在下一行。

```
mayfiecs@rocky ~ $ cd Desktop
mayfiecs@rocky ~/Desktop $ ls
Hello.java
mayfiecs@rocky ~/Desktop $ javac Hello.java
mayfiecs@rocky ~/Desktop $ ls
Hello.class  Hello.java
mayfiecs@rocky ~/Desktop $ java Hello
Hello, World!
mayfiecs@rocky ~/Desktop $
```

图 A-3：从命令行编译并运行 Hello.java

注意，执行命令 `javac` 时，必须指定一个*源代码文件名*（或多个用空格分隔的源代码文件名），而执行命令 `java` 时，只能指定单个*类名*。如果使用 DrJava，它会在幕后为你执行这些命令，并在 Interactions 窗格中显示输出。

花点时间学会这种高效而优雅的、与操作系统交互的方式，将提高你的生产率。不用命令行的人都不知道自己失去的是什么。

A.4　命令行测试

我们在 1.8 节中说过，相比于一次性编写所有的代码，逐步编写并调试代码的做法更有效。编写实现算法的代码后，通过测试来确定不管输入如何它都能正确地工作很重要。

本书很多地方都演示了程序测试技巧。大多数测试都基于简单的想法：程序的所做所为符合预期吗？对于简单的程序，多次运行并查看结果并不难；但反复输入相同的测试用例终将让你感到厌烦。

通过使用命令行，可将提供输入、比较期望输出与实际输出的过程自动化。其中的基本理念是将测试用例存储在纯文本文件中，并让 Java 以为这些输入来自键盘。实现这种想法的基本步骤如下。

(1) 确保能够编译并运行代码仓库 ThinkJavaCode 的目录 ch03 下的示例 `Convert.java`。

(2) 在 `Convert.java` 所在的目录中创建一个名为 `test.in` (in 表示输入）的纯文本文件，在其中输入如下内容并存盘：

```
193.04
```

(3) 再创建一个名为 `test.exp`（exp 表示预期）的纯文本文件，在其中输入如下内容并存盘：

```
193.04 cm = 6 ft, 4 in
```

(4) 打开一个终端窗口，并切换到这些文件所在的目录。执行下面的命令来测试这个程序：

```
java Convert < test.in > test.out
```

在命令行中，< 和 > 指的是*重定向运算符*（redirection operator）。前者将 test.in 的内容重定向到 System.in，使得就像通过键盘输入了它们一样。后者将 System.out 的内容重定向到新文件 test.out，这很像截屏。换而言之，文件 test.out 将包含程序的输出。

顺便说一句，完全可以在 DrJava（或其他开发环境）中对程序进行编译，再从命令行运行。熟悉这两种方法后，你就可根据工作需要选择合适的工具了。

现在可以对 test.out 和 test.exp 的内容进行比较了。如果这两个文件的内容相同，就说明程序的输出符合预期；如果不同，就说明程序存在 bug，而我们可根据实际输出对程序进行调试。所幸有一种通过命令行对文件进行比较的简单方式：

```
diff test.exp test.out
```

实用工具 `diff` 概述两个文件的差别。如果没有任何差别，`diff` 将不会显示任何内容，就这里而言，这正是我们希望的。如果期望输出与实际输出不同，我们就需要继续调试。通常而言，有问题的是程序，而 `diff` 让我们能够了解问题出在什么地方。但也可能程序没问题，而是预期输出不对。

`diff` 的输出可能不那么容易解读，好在有很多显示文件差异的图形工具。例如，在 Windows 系统中可使用 `WinMerge`；在 Mac 系统中可使用 `opendiff`（这是 Xcode 自带的）；在 Linux 系统中可使用 `meld`（如图 A-4 所示）。

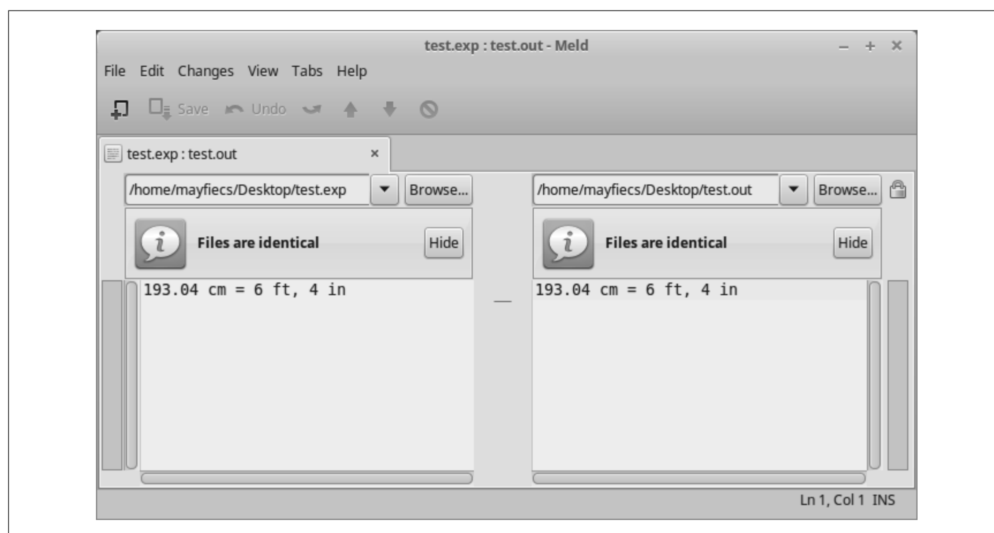


图 A-4：用 `meld` 比较预期输出和实际输出

不管用哪种工具，目标都是相同的：不断调试程序，直到实际输出与预期输出相同。

A.5　运行Checkstyle

Checkstyle 是一个命令行工具，可用于判断源代码是否遵循了指定的风格规则，还能检查出常见的编程错误，如类和方法设计中存在的问题。

要下载最新版 Checkstyle，可从 http://checkstyle.sourceforge.net/ 中下载相应的 JAR 文件。要运行 Checkstyle，可将这个 JAR 文件移到（或复制到）程序所在的目录，再打开一个终端窗口、切换到该目录并执行下面的命令：

```
java -jar checkstyle-*-all.jar -c /google_checks.xml *.java
```

其中的 `*` 是通配符（wildcard），分别指的是当前目录中的任何 Checkstyle 版本和所有 Java 源代码文件。该命令的输出指出了发现的每个问题所在的文件和行号；例如，下面的输出指的是文件 `Hello.java` 中始于第 93 行、第 5 列的一个方法：

```
Hello.java:93:5: Missing a Javadoc comment
```

文件 `/google_checks.xml` 位于下载的 JAR 文件中，其中包含 Google 的大部分风格规则。你可以使用 `/sun_checks.xml` 或提供自己的配置文件。更详细的信息请参阅 Checkstyle 网站。

只要经常用 Checkstyle 来检查编写的源代码，随着时间的推移，你很可能就会养成良好的风格习惯。然而，自动风格检查器的功能也存在局限。具体地说，对于注释的*质量*、变量名是否有*意义*以及算法的*结构*等，它们无法作出评估。

良好的注释能够让经验丰富的程序员更轻松地找出代码中的错误；好的变量名指出了程序的意图以及数据是如何组织的；而优秀的程序既高效又正确无误。

A.6　使用调试器进行跟踪

要想搞明白执行流程，包括形参和实参的工作原理，一种很不错的方式是使用*调试器*（debugger）。大多数调试器让你能够：

(1) 设置*断点*（breakpoint），即让程序执行到指定行后暂停。
(2) 以步进方式执行代码，即每次执行一行代码并查看每行代码的作用。
(3) 检查变量的值，看看它们在什么时候变了以及是怎么变的。

例如，在 DrJava 中打开一个程序，并将光标移到方法 `main` 的第一行，在按 Ctrl+B，这将在这行设置一个断点——显示为红色。按 Ctrl+Shift+D 将启用调试模式：窗口底部将出现一个新的窗格。如果你忘记了快捷键，调试器菜单也包含这些命令。

运行程序时，执行到第一个断点后将暂停，而调试窗格将显示*调用栈*（call stack），其中栈

顶为当前代码行所在的方法，如图 A-5 所示。你可能会惊讶地发现，在方法 `main` 之前竟然调用了如此多的方法！

调用栈的右边有多个按钮，让你能够以不同的步伐执行代码。还可以单击 Automatic Trace 按钮，让 DrJava 以每次一行的方式自动执行代码。

使用调试器犹如让计算机大声地审读代码。程序暂停后，可通过 Interactions 窗格检查甚至修改任何变量的值。

跟踪让你能够沿执行流程前行，并了解数据是如何在方法之间传递的。你可能预期代码会这样做，但调试器却指出它是那样做的。在这种情况下，你就知道代码可能存在什么样的错误。

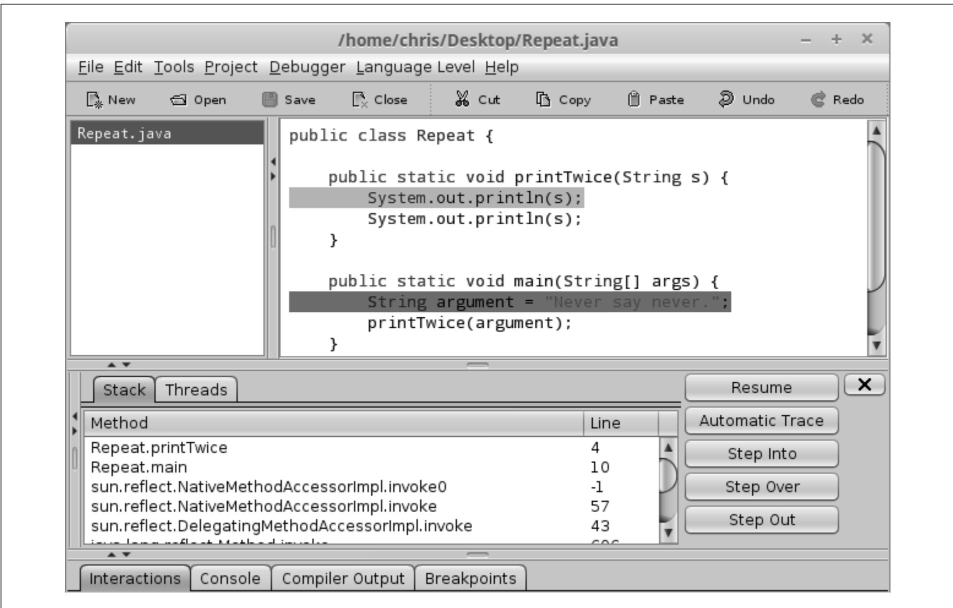


图 A-5：DrJava 调试器的截屏：程序执行到 `printTwice` 的第 1 行暂停，方法 `main` 的第 1 行有个断点

可在调试代码的同时对其进行编辑，但不建议这样做，因为如果在程序暂停时添加或删除多行代码，你可能根本不知道结果是怎么来的。

若想了解更多有关如何使用 DrJava 调试器的详细信息，请参阅 http://drjava.org/docs/user/ch09.html。

A.7 用JUnit进行测试

刚开始编写方法时，初学者通常会这样进行测试：在方法 `main` 中调用这些方法，并以人工方式检查结果。可能需要经常编写这样的代码，可用工具来简化这种工作。在知道正确结

果的情况下，可编写单元测试（unit test）来更好地执行测试工作。

例如，要测试 6.9 节的方法 `fibonacci`，可编写下面的代码：

```
public static void main(String[] args) {
    if (fibonacci(1) != 1) {
        System.err.println("fibonacci(1) is incorrect");
    }
    if (fibonacci(2) != 1) {
        System.err.println("fibonacci(2) is incorrect");
    }
    if (fibonacci(3) != 2) {
        System.err.println("fibonacci(3) is incorrect");
    }
}
```

这些测试代码的作用不言自明，但没必要这么长，可扩展性也不佳。另外，错误消息提供的信息也很有限。用单元测试框架可解决这些问题以及其他的问题。

JUnit 是一个常用的 Java 程序测试工具（参见 http://junit.org）。要使用它就必须创建包含测试方法的测试类：如果要测试的类名为 `Class`，则测试类应名为 `ClassTest`；如果类 `Class` 包含名为 `method` 的方法，`TestClass` 类应包含名为 `testMethod` 的方法。

例如，假设方法 `fibonacci` 属于类 `Series`，则相应的 JUnit 测试类和测试方法如下：

```
import junit.framework.TestCase;

public class SeriesTest extends TestCase {

    public void testFibonacci() {
        assertEquals(1, Series.fibonacci(1));
        assertEquals(1, Series.fibonacci(2));
        assertEquals(2, Series.fibonacci(3));
    }
}
```

这个示例使用了关键字 `extends`，这意味着新类 `SeriesTest` 继承了从包 `junit.framework` 中导入的既有类 `TestCase`。

很多开发环境能够自动生成测试类和测试方法。在 DrJava 中，可选择菜单 File>New JUnit Test Case 来生成空的测试类。

`assertEquals` 是由 `TestCase` 类提供的，它接受两个实参，并检查它们是否相等。如果相等，它就什么都不做；否则，就显示一条详细的错误消息。通常情况下，第一个实参为预期的值，即我们认为正确的值，而第二个实参是要检查的实际值。如果它们不相等，则未通过测试。

相比于自己编写 `if` 语句和 `System.err` 消息，用 `assertEquals` 更简洁。JUnit 还提供了其他

断言方法，如 `assertNull`、`assertSame` 和 `assertTrue`，它们可用于设计各种各样的测试。

要想在 DrJava 中直接运行 JUnit 测试，可单击工具栏中的 Test 按钮。如果所有的测试方法都通过了，右下角将出现一个绿条；否则 DrJava 将把你带到第一个失败的断言处。

A.8 术语表

- IDE
  集成开发环境，包含用于编辑、编译和调试程序的工具。

- JDK
  Java 开发包，包含编译器、Javadoc 和其他工具。

- JVM
  Java 虚拟机，对编译得到的字节码进行解释。

- 文本编辑器
  用于编辑大多数编程语言使用的纯文本文件的程序。

- JAR
  Java 归档文件，其实就是包含类和其他资源的 ZIP 文件。

- 命令行界面
  一种通过执行文本命令与计算机交互的途径。

- 重定向运算符
  一种用纯文本文件来替换 `System.in` 和 `System.out` 的命令行功能。

- 通配符
  一种让你能够用字符 * 来指定文件名模式的命令行功能。

- 调试器
  一种让你能够每次运行一条语句并查看变量值的工具。

- 断点
  一行代码，到这里后调试器将暂停执行程序。

- 调用栈
  有关方法调用以及每个方法返回后将在什么地方继续执行的历史记录。

- 单元测试
  执行程序中的单个方法，旨在测试其正确性和（或）效率的代码。

附录B

Java 2D图形

Java 库包含用于绘制 2D 图形的简单包 `java.awt`，其中的 AWT 指的是抽象窗口工具包（Abstract Window Toolkit）。这里只简要地介绍图形编程，更详细的信息请参阅相关的 Java 教程，网址为 https://docs.oracle.com/javase/tutorial/2d/。

B.1　创建图形

用 Java 创建图形的方式有很多种，其中最简单的方式是用 `java.awt.Canvas` 和 `java.awt.Graphics`。`Canvas` 是屏幕上的矩形空白区域，应用程序可在其中绘画；`Graphics` 类提供了基本的绘图方法，如 `drawLine`、`drawRect` 和 `drawString`。

下面的示例程序使用方法 `fillOval` 绘制一个圆：

```
import java.awt.Canvas;
import java.awt.Graphics;
import javax.swing.JFrame;

public class Drawing extends Canvas {

    public static void main(String[] args) {
        JFrame frame = new JFrame("My Drawing");
        Canvas canvas = new Drawing();
        canvas.setSize(400, 400);
        frame.add(canvas);
        frame.pack();
        frame.setVisible(true);
    }
```

```
        public void paint(Graphics g) {
            g.fillOval(100, 100, 200, 200);
        }
    }
```

这个 Drawing 类扩展了 Canvas，因此拥有 Canvas 提供的所有方法，包括 setSize。要了解其他方法，可参阅相关文档；而要找到这些文档，可在网上搜索 Java Canvas。

在方法 main 中，我们可以：

(1) 创建一个 JFrame 对象，这将是包含画布的窗口；
(2) 创建一个表示画布的 Drawing 对象，设置其宽度和高度，并将其加入到前面创建的框架中；
(3) 将框架的大小调整为与画布相同，再将其显示到屏幕上。

框架可见后，每当需要重绘画布时（如用户移动了窗口或调整了窗口的大小）都调用方法 paint。这个应用程序并不会在方法 main 返回时结束，而要等到 JFrame 关闭后才结束。如果运行这些代码，你将在灰色背景上看到一个黑色圆。

B.2　Graphics类的方法

你可能熟悉笛卡儿坐标（coordinate），其中 *x* 和 *y* 的值都可正可负。但在 Java 使用的坐标系中，原点位于左上角，因此 x 和 y 总是正整数。图 B-1 显示了这些坐标系。

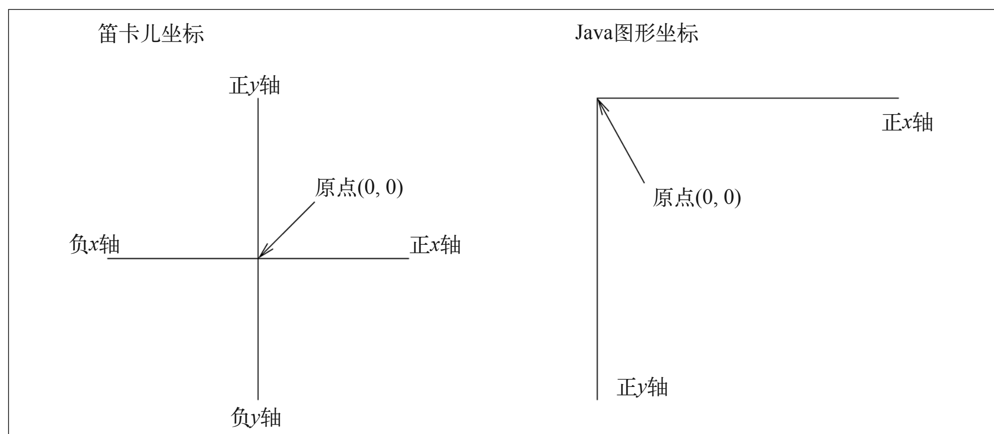


图 B-1：笛卡尔儿坐标和 Java 图形坐标的差别

图形坐标以像素（pixel）为单位，每个像素对应屏幕上的一点。

要想在画布上绘图，可对 Graphics 对象调用方法。无需创建这个 Graphics 对象，因为它将在创建 Canvas 对象时自动创建，并作为实参传递给方法 paint。

前面的示例使用了 fillOval，这个方法的特征标如下：

```
/**
 * 用当前颜色填充一个被指定矩形内接的椭圆
 */
public void fillOval(int x, int y, int width, int height)
```

其中的四个形参指定了将在其中绘制椭圆的*定界框*（bounding box）。x 和 y 指定定界框左上角的位置。定界框本身并不会显示出来，如图 B-2 所示。

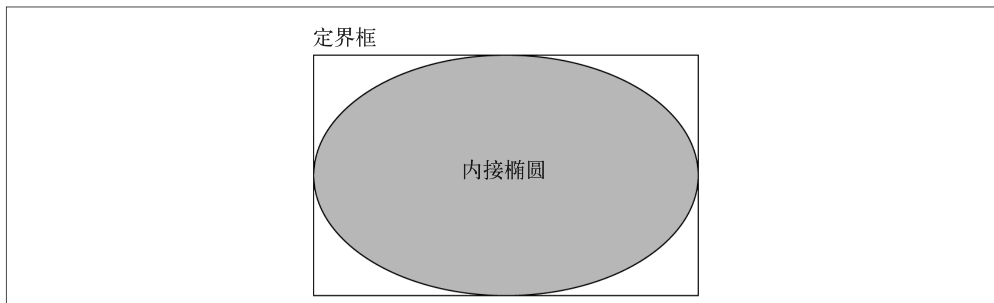


图 B-2：内接定界框的椭圆

要给形状指定颜色，可对 Graphics 对象调用方法 setColor：

```
g.setColor(Color.red);
```

方法 setColor 决定了随后绘制图形时将使用的颜色。Color.red 是 Color 类提供的一个常量，要使用它就必须导入 java.awt.Color。其他颜色常量包括：

```
black      blue     cyan    darkGray  gray   green
lightGray  magenta  orange  pink      white  yellow
```

还可通过指定红色、绿色和蓝色（GRB）分量来自定义颜色，如下所示：

```
Color purple = new Color(128, 0, 128);
```

每个颜色分量的可能取值都是 0（最暗）~255（最亮）的整数；(0, 0, 0) 表示黑色，而 (255, 255, 255) 表示白色。

可调用 setBackground 来设置 Canvas 的背景色：

```
canvas.setBackground(Color.white);
```

B.3 绘图示例

假设要绘制隐藏的米奇——表示米老鼠的图标（参见 https://en.wikipedia.org/wiki/Hidden_Mickey），可将前面绘制的椭圆作为脸，再加上两只耳朵。为提高代码的可读性，我们用

Rectangle 对象来表示定界框。

下面的方法接受一个 Rectangle 对象并调用 fillOval：

```
public void boxOval(Graphics g, Rectangle bb) {
    g.fillOval(bb.x, bb.y, bb.width, bb.height);
}
```

下面的方法绘制米老鼠：

```
public void mickey(Graphics g, Rectangle bb) {
    boxOval(g, bb);

    int dx = bb.width / 2;
    int dy = bb.height / 2;
    Rectangle half = new Rectangle(bb.x, bb.y, dx, dy);

    half.translate(-dx / 2, -dy / 2);
    boxOval(g, half);

    half.translate(dx * 2, 0);
    boxOval(g, half);
}
```

第 1 行绘制脸；接下来的 3 行创建一个较小的矩形。然后，我们将这个矩形向左上方平移，用作左耳的定界框，再将它向右平移，用作右耳的定界框。结果如图 B-3 所示。

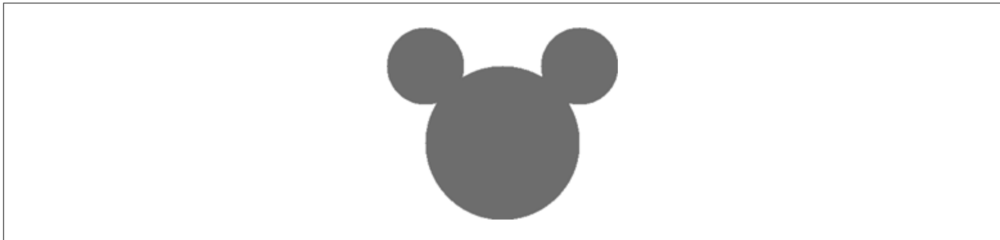


图 B-3：用 Java 图形绘制的隐藏的米奇

有关 Rectangle 和 translate 的更多详细信息，请参阅第 10 章。更多的绘图示例请参阅本附录末尾的练习。

B.4　术语表

- AWT

  抽象窗口工具包——用于创建图形用户界面的 Java 包。

- 坐标

  指定二维图形窗口中位置的值。

- *像素*
  坐标的度量单位。

- *定界框*
  一种指定矩形区域坐标的常见方式。

- RGB
  一种基于红、绿、蓝的颜色模型。

B.5 练习

本附录的示例代码位于仓库 ThinkJavaCode 的目录 ap02 中，有关如何下载这个仓库，请参阅前言中的“使用示例代码”一节。建议你先编译并运行这些示例，再动手做下面的练习。

练习B-1
绘制一面日本国旗：宽度比高度长的白色背景上有一个红色圆。

练习B-2
修改 Mickey.java，在每个耳朵上反复绘制两个耳朵，直到最小的耳朵的宽度只有 3 像素为止。

结果应类似于图 B-4 所示的米老鼠。*提示*：只需添加或修改几行代码就可以。

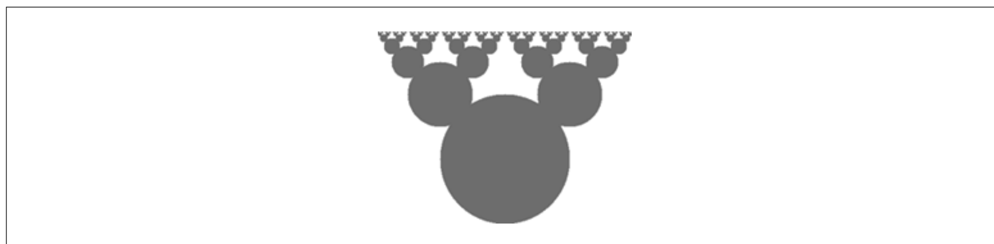


图 B-4：我们称之为“米老鼠”的递归形状

练习B-3
在这个练习中，你将绘制像在移动的摩尔纹，其中的相关原理请参阅 https://en.wikipedia.org/wiki/Moire_pattern。

(1) 在本书代码仓库的目录 app02 中，有一个名为 Moire.java 的文件。打开这个文件，并阅读其中的方法 `paint`。用草图描绘你希望它将绘制的图形，再运行它。结果与你预期的相同吗？
(2) 修改这个程序，增大或缩小相邻圆的间距，再看看结果。

(3) 修改这个程序，使其绘制以屏幕中点为圆心的同心圆，如图图 B-5（左）所示。相邻圆的间距必须足够小，这样才能出现摩尔干涉效果。

(4) 编写一个名为 `radial` 的方法，让它绘制一组如图 B-5（右）所示的辐射线，但要形成摩尔纹，这些线条的间距必须足够小。

(5) 几乎任何图案都能形成摩尔干涉效果。自己研究研究，看看你能创建哪些能够形成摩尔干涉效果的图案。

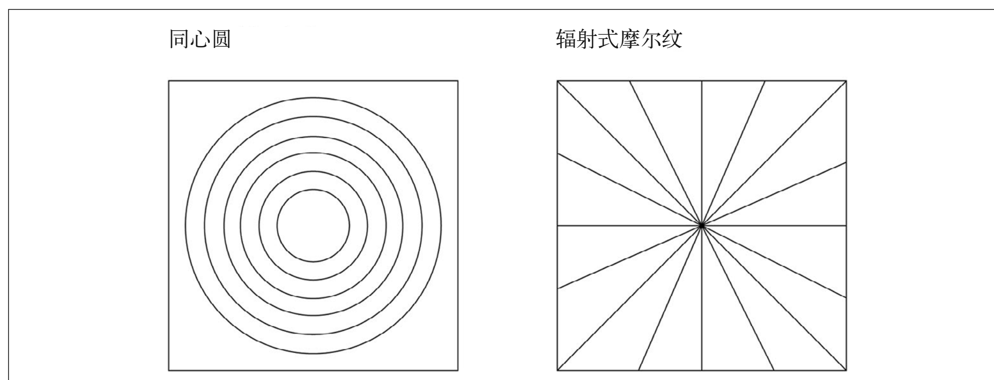


图 B-5：展示摩尔干涉的图案

附录 C

调试

虽然有关调试的建议贯穿本书，但我们认为将它们放在一个附录中会很有帮助。每当你在调试过程中陷入困境时，都应该重温这个附录。

哪种调试策略是最佳的呢？这取决于你面临的错误类型。

- 编译时错误：表明程序存在语法错误，如语句末尾遗漏了分号。
- 运行时错误：程序运行时出现的问题导致的错误，如无限递归导致 `StackOverflowError` 异常。
- 逻辑错误：导致程序的行为不正确，如表达式的计算顺序与你预期的不一致。

接下来的几节将介绍不同错误类型的相关调试技巧；有些技巧适用于特定类型的错误，对其他类型的错误可能不太管用。

C.1　编译时错误

最佳的调试方式是不用调试，因为你将错误扼杀在了摇篮中。为此，可采用 6.2 节中介绍的渐进开发，其中的关键是先编写一个可运行的简单程序，再每次添加少量的代码，这样发生错误时，你就会非常清楚它们出现在什么地方。

尽管如此，你可能还是会遭遇下面的情形。对于每种情形，我们都提供了一些处理建议。

C.1.1　编译器显示大量的错误消息

编译器显示 100 条错误消息时，并不意味着程序存在 100 个错误。编译器遇到错误时，通

常会暂时反应不过来。过了第一个错误后，它会尝试再次理清思路，但有时会错报错误。

只有第一条错误消息是确实可靠的。因此我们建议你每次只修复一个错误，并立即再次编译程序。你可能发现，添加一个分号或大括号就消除了 100 个错误。

C.1.2 编译器显示怪异的错误消息，怎么都消除不掉

首先，仔细阅读错误消息。错误消息可能包含简洁的专业术语，但通常隐藏着重要的信息。

即便没有提供其他信息，消息也至少指出了问题出现在程序的什么地方。实际上，它指出的是编译器阅读到什么地方时发现了问题，但错误并非一定就在那里。将编译器提供的信息作为参考，如果在编译器所说的地方没有发现错误，就扩大搜索范围。

错误通常出现在错误消息指出的位置的前面，但也可能出现在其他地方。例如，如果错误消息指出方法调用有问题，实际的错误很可能出现在方法定义中。

如果不能迅速找到错误，可先缓口气，再在整个程序中查找。为此，要先确保正确地缩进了程序代码，这样更容易发现语法错误。

然后，开始查找常见的语法错误。

(1) 检查所有的小括号和方括号都是成对的且嵌套正确。所有的方法定义都必须嵌套在类定义中。所有的程序语句必须位于方法定义中。
(2) 别忘了，Java 区分大小写。
(3) 检查语句末尾的分号。另外，大括号后面不需要分号。
(4) 确保代码中的所有字符串的引号都是成对的；确保使用了双引号来括起字符串，并使用了单引号来括起字符。
(5) 对于每条赋值语句而言，确保左边的类型与右边的类型相同。确保表达式的左边是变量名或其他可赋值的东西（如数组元素）。
(6) 确保每个方法调用指定的实参的类型和排列顺序是正确的，且用来调用方法的对象的类型也是正确的。
(7) 调用值方法时，确保使用了它返回的结果；调用 `void` 方法时，确保没有使用它返回的结果。
(8) 调用实例方法时，确保调用它的对象的类型是正确的；在类外面调用其静态方法时，确保用句点表示法指定了类名。
(9) 在实例方法中，可在没有指定对象的情况下引用实例变量。如果你在静态方法中这样做（无论是否使用 `this`），将出现类似于下面的错误消息：non-static variable `x` cannot be referenced from a static context（在静态方法中不能引用非静态变量 `x`）。

如果还是没有找到错误，请进入下一节。

C1.3　怎么做都无法让程序通过编译

如果编译器说存在错误，但你找不出来，可能是因为你和编译器看的代码不同。请检查开发环境，确保你编辑的程序就是编译器编译的程序。

程序有多个版本时，经常会出现这样的情况。你编辑的是一个版本，而编译的是另一个版本。

如果你不确定是不是这样的，可在程序开头故意添加一个显而易见的语法错误，再重新编译。如果编译器没有发现新添的错误，就很可能是开发环境的设置有问题。

如果你仔细研究了代码，并确定编译器编译的是正确的源代码文件，就该拿出杀手锏了：二分调试（debugging by bisection）。

- 备份当前调试的文件。如果调试的是 Bob.java，就创建副本并将其命名为 Bob.java.old。
- 将 Bob.java 的代码删除约一半，再重新编译。
  - 如果程序能够通过编译，就说明错误发生在被删除的代码中。将删除的代码恢复一半，并再次进行编译。
  - 如果程序不能通过编译，那么错误肯定出在刚恢复的代码中。将刚恢复的代码删除一半，并再次进行编译。
- 找到并修复错误后，再逐步恢复被删除的代码。

这种做法一点都不优雅，但找到错误的速度可能比你想象得快，而且非常可靠。也适用于其他编程语言！

C.1.4　按编译器说的做了，但还是不管用

有些错误消息还包含修复建议，如“`Golfer` 类必须声明为抽象的，它没有定义接口 `java.lang.Comparable` 中的方法 `int compareTo(java.lang.Object)`”。这让你以为编译器要求你将 `Golfer` 声明为抽象类；如果你还处于阅读本书的水平，很可能不知道抽象类是什么，也不知道该如何声明为抽象类。

好在编译器错了。对于这里的错误，解决方案是确保 `Golfer` 定义了将 `Object` 作为参数的方法 `compareTo`。

可别让编译器牵着鼻子走。错误消息表明存在错误，但推荐的解决之道并不可靠。

C.2　运行时错误

导致运行时错误的原因并非总是那么明显，但在程序中添加打印语句通常能找出原因。

C.2.1 程序挂起

如果程序停止运行，看起来什么都不做，我们就说它“挂起”了。这通常意味着遭遇了无限循环或无限递归。

- 如果你怀疑问题出在某个循环上，可在该循环前面添加一条显示“进入循环”的打印语句，并在它后面添加一条显示“退出循环”的打印语句。

  运行程序。如果你看到了第一条消息，但没有看到第二条，就知道程序在什么地方卡壳了。为解决这种问题，请参阅“无限循环”一节。

- 在大多数情况下，无限递归会导致程序运行一段时间后出现 `StackOverflowError` 异常。如果出现这种异常，请参阅“无限递归”一节。

  如果没有出现 `StackOverflowError` 异常，但你怀疑问题出在某个递归方法上，可用“无限递归”一节介绍的技巧来解决问题。

- 如果上述两个建议都不管用，可能是因为你没有搞明白程序的执行流程。在这种情况下，可参阅“执行流程”一节。

1. 无限循环

如果你认为程序包含无限循环并知道是哪一个，可在这个循环末尾添加打印语句，以显示条件变量的值以及条件的值。

例如：

```
while (x > 0 && y < 0) {
    // 修改x
    // 修改y

    System.out.println("x: " + x);
    System.out.println("y: " + y);
    System.out.println("condition: " + (x > 0 && y < 0));
}
```

这样的话，当运行程序时，该循环每执行一次都将显示 3 行输出。最后一次执行循环时，添加应为 `false`。如果循环不断地执行，你将看到 x 和 y 的值，进而也许能够搞明白它们未能正确更新的原因。

2. 无限递归

在大多数情况下，无限递归将导致程序引发 `StackOverflowError` 异常。但如果程序的运行速度很慢，可能需要很长时间才能填满栈。

如果你知道无限递归是哪个方法导致的，可检查它是否包含基线条件。必须存在某种条件，让方法不再进行递归调用而是返回。如果没有，就需要重新审视算法并找出基线条件。

如果有基线条件，但程序好像满足不了这个条件，可在方法开头添加显示形参的打印语句。这样的话，在程序运行期间，每当这个方法被调用时，你都将看到几行输出并获悉形参的值。如果形参没有逐渐接近基线条件，你也许能够搞明白其中的原因。

3. 执行流程

如果不知道程序的执行流程，可在每个方法开头添加打印语句，以显示“进入方法 `foo`”这样的消息，其中 `foo` 为当前方法的名称。这样的话，程序运行时，每个被调用的方法都将留下痕迹。

还可显示每个方法收到的实参。这样的话，你可以在程序运行时检查实参的值是否合理，还能发现最常见的错误之一——实参的指定顺序不正确。

C.2.2　程序运行时出现异常

出现异常时，Java 会显示一条消息，其中包含异常的名称、出现异常的代码的行号以及“栈跟踪”。栈跟踪包含当时运行的方法和方法调用链。

你应该先检查错误发生的地方，并看看能不能找出其中的原因。

- `NullPointerException`

  试图通过值为 `null` 的对象变量访问实例变量或调用方法时，将引发这种异常。在这种情况下，你需要确定哪个变量为 `null`，再搞清楚它是怎么变成 `null` 的。

  别忘了，声明数组变量时，其元素在被赋值前默认为 `null`。例如，下面的代码将引发 `NullPointerException` 异常：

  ```
  int[] array = new Point[5];
  System.out.println(array[0].x);
  ```

- `ArrayIndexOutOfBoundsException`

  访问数组时，如果使用的索引为负或大于 `array.length - 1`，将引发这种异常。如果你能够确定问题出在什么地方，可在它前面添加打印语句来显示索引的值和数组的长度。数组的长度对吗？索引的值对吗？

  然后，在程序中往后回溯，确定数组和索引来自何方。找到最近的赋值语句，看看其所作所为是否正确。如果数组或索引为形参，那么就跳转到调用方法的地方，看看这些值来自何方。

- `StackOverflowError`

  参见前面的“无限递归”一节。

- `FileNotFoundException`

  这意味着 Java 没有找到要查找的文件。如果你使用的是基于项目的开发环境，如 Eclipse，可能必须将这个文件导入项目。否则，请确保这个文件存在且路径正确。这种问题与文件系统相关，可能难以追查。

- `ArithmeticException`

  算术运算的执行出现了问题，如除以零。

C.2.3　添加了很多打印语句，输出都泛滥成灾了

用打印语句帮助调试带来的一个问题是，最终的输出可能泛滥成灾。解决之道有两个：要么简化输出，要么简化程序。

要想简化输出，可将不再有帮助的打印语句删除或注释掉、合并打印语句或设置输出的格式使其易于理解。开发程序时，应编写代码来生成简洁而信息丰富的消息，对程序的所作所为进行跟踪。

要想简化程序，可缩小程序处理的问题的规模。例如，对数组进行排序时，可使用较小的数组。如果程序从用户那里获取输入，可向它提供导致错误的最简单输入。

另外，对代码进行清理：删除多余或实验性部分，并重新组织程序使其更易阅读。例如，如果你怀疑错误出在程序中的一个多层嵌套的部分，可用更简单的结构重新编写这部分；如果你怀疑错误出现在一个很大的方法中，可将这个方法分成多个小方法，再分别进行测试。

在确定最简单的测试用例的过程中，常常能够发现导致 bug 的线索。例如，如果你发现程序在数组包含偶数个元素时没问题，但包含奇数个元素时出现问题，这可能就获得了找出原因的线索。

重新组织程序可帮助你找出微妙的 bug。如果修改程序时发现，原本以为这样的修改不会有任何影响，但结果并非如此，这便透露出了蛛丝马迹。

C.3　逻辑错误

C.3.1　程序不管用

逻辑错误难以发现，因为编译器和解释器不会提供有关这种错误的任何信息。只有知道程序该如何做时，你才能知道程序没有这样做。

首先，你需要在代码和程序的实际行为之间建立联系。你需要就程序的实际行为作出假

设。下面是你需要回答的一些问题。

- 存在程序该做却没有做的事情吗？找出执行这项功能的代码片段，确定它在你认为该执行的时候执行了。参见前面的“执行流程”一节。
- 出现了原本不该发生的事情吗？在程序中找到执行这项功能的代码，看看它是不是在不该执行的时候执行了。
- 是否存在带来意外影响的代码片段？确保你搞明白了这些代码，尤其是调用了 Java 库中的方法时。阅读这些方法的相关文档，并用简单的测试用例来尝试使用它们。它们的功能可能不是你想的那样。

要编写程序，你需要建立有关代码行为的心理模型。如果代码的行为不符合预期，有问题的可能不是程序，而是你的心理模型。

要想校正你的心理模型，最佳的方式是将程序分成多个部分（通常是类和方法），再分别测试它们。一旦找出心理模型与实际情况的偏差，你就能把问题给解决了。

下面是需要检查的一些常见逻辑错误。

- 别忘了，整数除法的结果总是向下圆整的。如果你要获得小数结果，应用 `double` 执行除法运算。推而广之，用整数表示可数的东西；用浮点数表示不可数的东西。
- 浮点数只是近似值，不要指望它们绝对精确。根本就不能用运算符 `==` 来比较浮点数。换句话说，不要编写 `if(d == 1.23)` 这样的代码，而应编写 `if(Math.abs(d - 1.23) < .000001)` 这样的代码。
- 用于对象时，相等运算符（`==`）检查两个对象是否相同。如果你要比较两个对象是否相等，应使用方法 `equals`。
- 对于用户定义的类型，默认的 `equals` 方法检查两个对象是否相同。如果你要赋予相等不同的含义，必须重写这个方法。
- 继承可能带来微妙的逻辑错误，因为你可能在不知不觉间运行了继承而来的代码。请参阅前面的“执行流程”一节。

C.3.2 冗长表达式的结果出乎意料

你完全可以编写复杂的表达式，只要它们易于理解，但调试起来可能很麻烦。通常而言，最好将复杂表达式分解成一系列给临时变量赋值的赋值语句。

```
rect.setLocation(rect.getLocation().translate(
                 -rect.getWidth(), -rect.getHeight()));
```

前面的示例可重写为下面这样：

```
int dx = -rect.getWidth();
int dy = -rect.getHeight();
```

```
Point location = rect.getLocation();
Point newLocation = location.translate(dx, dy);
rect.setLocation(newLocation);
```

第二个版本更容易理解，这要部分归功于变量名提供了额外的说明。这个版本调试起来也更容易，因为你可以检查临时变量的类型并显示它们的值。

冗长表达式可能存在的另一个问题是，运算的执行顺序可能并非你以为的那样。例如，为计算 $x/(2\pi)$，你可能编写下面的代码：

```
double y = x / 2 * Math.PI;
```

这不对，因为乘法和除法运算的优先级相同，因此按从左到右的顺序执行。上述代码计算的是 x 除以 2 再乘 π。

如果你对运算顺序没有把握，可查看相关文档，也可用括号来明确地指定。

```
double y = x / (2 * Math.PI);
```

这个版本是正确的，对不记得运算顺序的人来说也更容易理解。

C.3.3　方法的返回值出乎意料

如果你在返回语句中包含复杂的表达式，就根本没有机会在返回前显示这个表达式的值。

```
public Rectangle intersection(Rectangle a, Rectangle b) {
    return new Rectangle(
        Math.min(a.x, b.x), Math.min(a.y, b.y),
        Math.max(a.x + a.width, b.x + b.width)
            - Math.min(a.x, b.x)
        Math.max(a.y + a.height, b.y + b.height)
            - Math.min(a.y, b.y));
}
```

不应将整个表达式放在一条语句中，而应使用一系列临时变量：

```
public Rectangle intersection(Rectangle a, Rectangle b) {
    int x1 = Math.min(a.x, b.x);
    int y2 = Math.min(a.y, b.y);
    int x2 = Math.max(a.x + a.width, b.x + b.width);
    int y2 = Math.max(a.y + a.height, b.y + b.height);
    Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);
    return rect;
}
```

这样就可以在返回前显示任何中间变量的值了。另外，通过重用 `x1` 和 `y1`，代码也更短了。

C.3.4　打印语句什么都不显示

如果你使用的是方法 `println`，输出将立即显示出来，但如果你使用的是 `print`，输出将被存储起来，直到出现换行符才显示出来（至少在有些环境中如此）。如果程序直到终止都没有显示换行符，你可能根本看不到存储的输出。如果你怀疑这就是罪魁祸首，可将部分或全部 `print` 语句改为 `println` 语句。

C.3.5　陷入了绝境，无法自拔

首先，离开计算机一会儿。计算机发射的电波会影响人的大脑，让人出现如下症状：

- 气馁和愤怒；
- 怪异的想法（“计算机讨厌我。”）和迷信（“这个程序只在我将帽子反戴时才能正确地运行。”）；
- 酸葡萄心理（“这个程序真不怎样。”）。

如果你出现了上述任何症状，赶快起来走一走。冷静下来后再来研究程序。程序当前的行为是什么样的？导致这种行为的原因可能是什么？程序最后一次正确地运行之后，你都做了些什么？

找出有些 bug 就是需要时间。人在放松时常常容易找出 bug，如坐公交车、洗澡和躺在床上时。

C.3.6　必须得有人帮我

每个人都会遇到这样的情况，即便是最优秀的程序员，也有陷入困境的时候。有时候，你需要别人的帮助。

找人帮忙前，务必尝试本附录介绍的所有方法。

你的程序应尽可能简单，并使用尽可能简单的输入来引发错误；你应在合适的地方添加打印语句，且这些语句的输出应易于理解；你对问题有足够的了解，能够简练地进行描述。

帮忙的人来了后，向他们提供所需的信息。

- bug 是什么类型的？编译时错误、运行时错误还是逻辑错误？
- 这种错误发生前，你做了什么？你最后编写的是哪些代码行？或者哪个测试用例未通过？
- 如果错误发生在编译时或运行时，显示的是什么错误消息？它指出程序的什么地方有问题？
- 你采取了哪些措施？得出了什么样的结论？

等你向人说明完问题时，你可能已经找到了答案。鉴于这种现象非常普遍，有人推荐使用“橡皮鸭调试法”。这种调试法的步骤如下。

(1) 买个标准款橡皮鸭。
(2) 当你面对问题无计可施时，将橡皮鸭放在前面的桌子上，并对它说，“橡皮鸭，我深陷困境，情况是这样的……”。
(3) 向橡皮鸭描述面临的问题。
(4) 发现解决方案。
(5) 向橡皮鸭致谢。

没跟你开玩笑，这真的管用！详情见 https://en.wikipedia.org/wiki/Rubber-duck-debugging。

C.3.7　终于找到bug了！

bug 找到后，如何修复通常来说是显而易见的，但并非总是如此。有些看起来是 bug 的东西其实表明你没有理解程序或你使用的算法有问题。在这种情况下，你可能需要重新审视算法或调整心理模型。可暂时离开计算机，理清思路、手工执行测试用例或绘制计算图。

修复 bug 后，不要立即投入到再造新 bug 的编程过程中。花点时间想想这是什么样的 bug、你为何会犯这样的错误、这种错误有何特征以及如何更快地找出它。这样的话，再遇到类似的情况时，你就能更快找到 bug，乃至再也不让这样的 bug 出现。

作者简介

Allen B. Downey 是欧林学院的计算机教授。曾任教于韦尔斯利女子学院、科尔比学院和加州大学伯克利分校，拥有加州大学伯克利分校计算机博士学位以及麻省理工学院学士和硕士学位。

Chris Mayfield 是詹姆斯麦迪逊大学的计算机助理教授，致力于计算机教育和职业发展的研究；拥有普渡大学计算机博士学位以及犹他大学计算机和德语学士学位。

封面简介

本书封面上的动物为红尾黑凤头鹦鹉，也叫班克斯黑凤头鹦鹉——这是以 18 世纪的英国植物学家约瑟夫 · 班克斯命名的。这种大型鸟类分布在澳大利亚，栖息于森林、草原和河岸等众多地方，通常在桉树上筑巢。

顾名思义，红尾黑凤头鹦鹉的羽毛为黑色，只有雄鸟的尾部有鲜艳的红色斑纹。这种鸟类长约 2 英尺，重 1~2 磅。与其他黑凤头鹦鹉科鸟类一样，红尾黑凤头鹦鹉的喙庞大并呈流线型，头上有羽冠，脚上有四个脚趾——两个向前、两个向后，因此能够用一只脚抓住树枝，另一只脚抓住其他物体。有趣的是，红尾黑凤头鹦鹉大多惯用左脚。

红尾黑凤头鹦鹉主要以桉树的种子为食，偶尔也吃些坚果、水果、昆虫和谷物。这种鸟类会成群地在食物丰富的地方飞翔，发出的声音极大，但在人前通常非常胆怯。

红尾黑凤头鹦鹉在森林中筑巢和觅食，因此易受滥砍滥伐的影响；在澳大利亚东南部，滥砍滥伐已经让一些红尾黑凤头鹦鹉面临生存威胁。另外，虽然在澳大利亚饲养红尾黑凤头鹦鹉必须有专门的许可证，但它们依然受到非法走私的威胁。因为这种鸟类被圈养时能够活很长时间，所以需求非常旺盛。

O'Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

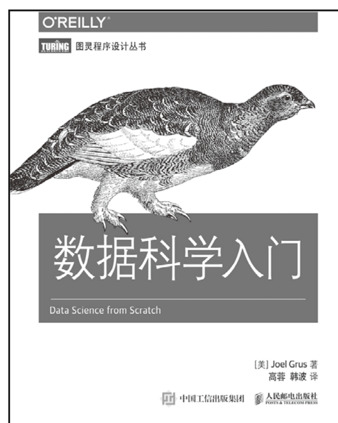
封面图片来自 *Wood's Illustrated Natural History* 一书。

延 展 阅 读



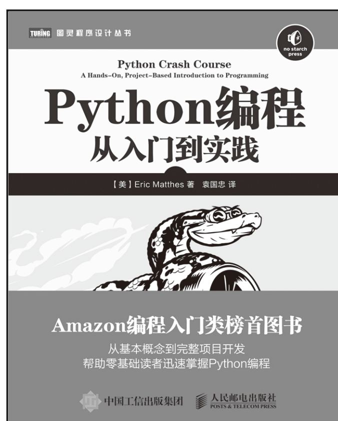
程序员版《人性的弱点》，提升职业生涯软技能

书号：978-7-115-43418-0
定价：45.00 元



介绍数据科学基本知识的重量级读本，Google 数据科学家作品

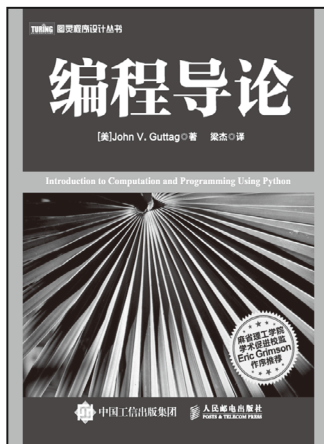
书号：978-7-115-41741-1
定价：69.00 元



Amazon 编程入门类榜首图书
从基本概念到完整项目开发，帮助零基础读者迅速掌握 Python 编程

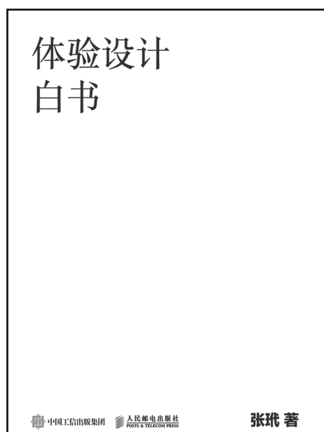
书号：978-7-115-42802-8
定价：89.00 元

延 展 阅 读



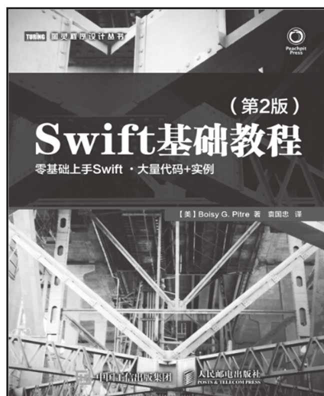
麻省理工学院开放式课程最受欢迎的计算机课程的教材

书号：978-7-115-38801-8
定价：59.00 元



从零了解体验设计，提升思辨能力以及设计能力，做出用户以及目标客户都满意的产品

书号：978-7-115-42709-0
定价：49.00 元



零基础上手 Swift
大量代码 + 实例

书号：978-7-115-42230-9
定价：49.00 元

O'REILLY®

Java编程思维

本书是一本实用的计算机入门教程，目前已被众多大学和高中选作教材。书中不仅介绍Java编程，还阐明如何养成计算机科学家才具备的思维方式，让读者学会怎样将编程作为实现目的的手段。

作者从最基本的概念着手，逐步转入更复杂的主题，如递归和面向对象编程。每一章都简明扼要，章末附有练习，可将学到的知识立即付诸实践。

- 每次学习一个概念：复杂的主题分为多个部分，并辅以示例进行讲解。
- 学会如何明确地描述问题、如何创造性地寻找解决方案以及如何编写清晰准确的程序。
- 寻找最适合的开发方式，锻炼最重要的调试技能。
- 了解输入和输出、决策和循环、类和方法、字符串和数组之间的关系。
- 填字游戏、图形、拼图和扑克牌方面的程序开发练习。

Allen B. Downey，欧林学院计算机教授，曾任教于韦尔斯利女子学院、科尔比学院和加州大学伯克利分校。拥有加州大学伯克利分校计算机博士学位以及麻省理工学院学士和硕士学位。

Chris Mayfield，詹姆斯麦迪逊大学计算机助理教授，致力于计算机教育和职业发展的研究。拥有普渡大学计算机博士学位以及犹他大学计算机和德语学士学位。

“这本书专注于解决问题，不仅介绍了如何编程，还深入探讨了计算机科学背后的概念，是引导初学者养成计算机科学思维的佳作。”

——Rebecca Dovi
CodeVA教育总监

JAVA/BEGINNING PROGRAMMING

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-44015-0
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks